The Python Library Reference

Version 3.14.0a5

Guido van Rossum and the Python development team

février 17, 2025

Python Software Foundation
Email : docs@python.org

Table des matières

1 Introduction **3**
1.1 Notes sur la disponibilité . . . 3
1.1.1 Plateformes WebAssembly . . . 4
1.1.2 Mobile platforms . . . 4

2 Fonctions natives **7**

3 Constantes natives **35**
3.1 Constantes ajoutées par le module `site` . . . 36

4 Types natifs **37**
4.1 Valeurs booléennes . . . 37
4.2 Opérations booléennes — `and`, `or`, `not` . . . 37
4.3 Comparaisons . . . 38
4.4 Types numériques — `int`, `float`, `complex` . . . 38
4.4.1 Opérations sur les bits des nombres entiers . . . 40
4.4.2 Méthodes supplémentaires sur les entiers . . . 41
4.4.3 Méthodes supplémentaires sur les nombres à virgule flottante . . . 43
4.4.4 Additional Methods on Complex . . . 44
4.4.5 Hachage des types numériques . . . 44
4.5 Boolean Type - `bool` . . . 46
4.6 Les types itérateurs . . . 46
4.6.1 Types générateurs . . . 47
4.7 Types séquentiels — `list`, `tuple`, `range` . . . 47
4.7.1 Opérations communes sur les séquences . . . 47
4.7.2 Types de séquences immuables . . . 48
4.7.3 Types de séquences mutables . . . 49
4.7.4 Listes . . . 49
4.7.5 *N*-uplets . . . 50
4.7.6 *Ranges* . . . 51
4.8 Text and Binary Sequence Type Methods Summary . . . 52
4.9 Type Séquence de Texte — `str` . . . 53
4.9.1 Méthodes de chaînes de caractères . . . 54
4.9.2 Formatage de chaines à la `printf` . . . 63
4.10 Séquences Binaires — `bytes`, `bytearray`, `vue mémoire` . . . 65
4.10.1 Objets *bytes* . . . 65
4.10.2 Objets *bytearray* . . . 66
4.10.3 Opérations sur les *bytes* et *bytearray* . . . 67
4.10.4 Formatage de *bytes* a la `printf` . . . 79
4.10.5 Vues mémoire . . . 81
4.11 Types d'ensembles — `set`, `frozenset` . . . 88

4.12 Les types de correspondances — `dict` . . . 90
4.12.1 Les vues de dictionnaires . . . 94
4.13 Le type gestionnaire de contexte . . . 95
4.14 Types d'annotation de type — Alias générique, Union . . . 96
4.14.1 Type Alias générique . . . 96
4.14.2 Type Union . . . 100
4.15 Autres types natifs . . . 102
4.15.1 Modules . . . 102
4.15.2 Les classes et instances de classes . . . 102
4.15.3 Fonctions . . . 102
4.15.4 Méthodes . . . 102
4.15.5 Objets code . . . 103
4.15.6 Objets type . . . 103
4.15.7 L'objet Null . . . 103
4.15.8 L'objet points de suspension (ou ellipse) . . . 103
4.15.9 L'objet *NotImplemented* . . . 103
4.15.10 Objets internes . . . 103
4.16 Attributs spéciaux . . . 103
4.17 Limitation de longueur de conversion de chaîne vers un entier . . . 104
4.17.1 API concernées . . . 105
4.17.2 Configuration de la limite . . . 105
4.17.3 Configuration recommandée . . . 106

5 Exceptions natives **107**
5.1 Contexte des exceptions . . . 107
5.2 Hériter des exceptions natives . . . 108
5.3 Classes mères . . . 108
5.4 Exceptions concrètes . . . 109
5.4.1 Exceptions système . . . 115
5.5 Avertissements . . . 116
5.6 Exception groups . . . 117
5.7 Hiérarchie des exceptions . . . 119

6 Services de Manipulation de Texte **121**
6.1 `string` --- Common string operations . . . 121
6.1.1 Chaînes constantes . . . 121
6.1.2 Formatage personnalisé de chaîne . . . 122
6.1.3 Syntaxe de formatage de chaîne . . . 123
6.1.4 Chaînes modèles . . . 130
6.1.5 Fonctions d'assistance . . . 132
6.2 `re` --- Regular expression operations . . . 132
6.2.1 Syntaxe des expressions rationnelles . . . 133
6.2.2 Contenu du module . . . 140
6.2.3 Objets d'expressions rationnelles . . . 146
6.2.4 Objets de correspondance . . . 147
6.2.5 Exemples d'expressions rationnelles . . . 150
6.3 `difflib` --- Helpers for computing deltas . . . 155
6.3.1 SequenceMatcher Objects . . . 160
6.3.2 SequenceMatcher Examples . . . 163
6.3.3 Differ Objects . . . 163
6.3.4 Differ Example . . . 164
6.3.5 A command-line interface to difflib . . . 165
6.3.6 ndiff example . . . 166
6.4 `textwrap` --- Text wrapping and filling . . . 168
6.5 `unicodedata` --- Unicode Database . . . 172
6.6 `stringprep` --- Internet String Preparation . . . 174
6.7 `readline` --- GNU readline interface . . . 175
6.7.1 Fichier d'initialisation . . . 176

6.7.2 Tampon de ligne . . . 176
6.7.3 Fichier d'historique . . . 176
6.7.4 Liste d'historique . . . 177
6.7.5 Fonctions de rappel au démarrage . . . 177
6.7.6 Complétion . . . 178
6.7.7 Exemple . . . 178
6.8 `rlcompleter` --- Completion function for GNU readline . . . 180

7 Services autour des Données Binaires **181**
7.1 `struct` --- Interpret bytes as packed binary data . . . 181
7.1.1 Fonctions et exceptions . . . 182
7.1.2 Chaînes de spécification du format . . . 182
7.1.3 Applications . . . 187
7.1.4 Classes . . . 188
7.2 `codecs` --- Codec registry and base classes . . . 189
7.2.1 Classes de base de codecs . . . 192
7.2.2 Encodings and Unicode . . . 198
7.2.3 Standard Encodings . . . 199
7.2.4 Python Specific Encodings . . . 202
7.2.5 `encodings.idna` --- Internationalized Domain Names in Applications . . . 204
7.2.6 `encodings.mbcs` --- Windows ANSI codepage . . . 205
7.2.7 `encodings.utf_8_sig` --- UTF-8 codec with BOM signature . . . 205

8 Types de données **207**
8.1 `datetime` — Types de base pour la date et l'heure . . . 207
8.1.1 Objets avisés et naïfs . . . 208
8.1.2 Constantes . . . 208
8.1.3 Types disponibles . . . 208
8.1.4 Objets `timedelta` . . . 209
8.1.5 Objets `date` . . . 213
8.1.6 Objets `datetime` . . . 218
8.1.7 Objets `time` . . . 229
8.1.8 Objets `tzinfo` . . . 233
8.1.9 Objets `timezone` . . . 239
8.1.10 `strftime()` and `strptime()` Behavior . . . 240
8.2 `zoneinfo` --- IANA time zone support . . . 245
8.2.1 Utilisation de `ZoneInfo` . . . 245
8.2.2 Sources de données . . . 246
8.2.3 La classe `ZoneInfo` . . . 247
8.2.4 Fonctions . . . 249
8.2.5 Variables globales . . . 249
8.2.6 Exceptions et avertissements . . . 250
8.3 `calendar` --- General calendar-related functions . . . 250
8.3.1 Command-line usage . . . 257
8.4 `collections` --- Container datatypes . . . 258
8.4.1 Objets `ChainMap` . . . 259
8.4.2 Objets `Counter` . . . 261
8.4.3 Objets `deque` . . . 264
8.4.4 Objets `defaultdict` . . . 268
8.4.5 `namedtuple()` : fonction de construction pour *n*-uplets avec des champs nommés . . . 269
8.4.6 Objets `OrderedDict` . . . 273
8.4.7 Objets `UserDict` . . . 275
8.4.8 Objets `UserList` . . . 276
8.4.9 Objets `UserString` . . . 276
8.5 `collections.abc` --- Abstract Base Classes for Containers . . . 276
8.5.1 Classes de base abstraites de collections . . . 278
8.5.2 Collections Abstract Base Classes -- Detailed Descriptions . . . 279
8.5.3 Exemples et recettes . . . 281

8.6 heapq --- Heap queue algorithm . . . 282
8.6.1 Exemples simples . . . 283
8.6.2 Notes d'implémentation de la file de priorité . . . 283
8.6.3 Théorie . . . 284
8.7 bisect --- Array bisection algorithm . . . 286
8.7.1 Notes sur la performance . . . 287
8.7.2 Chercher dans des listes triées . . . 287
8.7.3 Exemples . . . 288
8.8 array --- Efficient arrays of numeric values . . . 289
8.9 weakref --- Weak references . . . 292
8.9.1 Objets à références faibles . . . 296
8.9.2 Exemple . . . 297
8.9.3 Finalizer Objects . . . 297
8.9.4 Comparing finalizers with __del__() methods . . . 298
8.10 types --- Dynamic type creation and names for built-in types . . . 299
8.10.1 Dynamic Type Creation . . . 300
8.10.2 Standard Interpreter Types . . . 301
8.10.3 Additional Utility Classes and Functions . . . 304
8.10.4 Coroutine Utility Functions . . . 305
8.11 copy --- Shallow and deep copy operations . . . 306
8.12 pprint — L'affichage élégant de données . . . 307
8.12.1 Fonctions . . . 307
8.12.2 Les Objets PrettyPrinter . . . 308
8.12.3 Exemple . . . 310
8.13 reprlib --- Alternate repr() implementation . . . 313
8.13.1 Repr Objects . . . 314
8.13.2 Subclassing Repr Objects . . . 315
8.14 enum --- Support for enumerations . . . 316
8.14.1 Contenu du module . . . 316
8.14.2 Types de données . . . 318
8.14.3 Utilitaires et décorateurs . . . 329
8.14.4 Notes . . . 330
8.15 graphlib --- Functionality to operate with graph-like structures . . . 331
8.15.1 Exceptions . . . 333

9 Modules numériques et mathématiques 335
9.1 numbers --- Numeric abstract base classes . . . 335
9.1.1 La tour numérique . . . 335
9.1.2 Notes for type implementers . . . 336
9.2 math --- Mathematical functions . . . 338
9.2.1 Number-theoretic functions . . . 340
9.2.2 Floating point arithmetic . . . 340
9.2.3 Floating point manipulation functions . . . 341
9.2.4 Power, exponential and logarithmic functions . . . 343
9.2.5 Summation and product functions . . . 344
9.2.6 Conversion angulaire . . . 345
9.2.7 Fonctions trigonométriques . . . 345
9.2.8 Fonctions hyperboliques . . . 345
9.2.9 Fonctions spéciales . . . 346
9.2.10 Constantes . . . 346
9.3 cmath --- Mathematical functions for complex numbers . . . 347
9.3.1 Conversion vers et à partir de coordonnées polaires . . . 348
9.3.2 Fonctions logarithme et exponentielle . . . 348
9.3.3 Fonctions trigonométriques . . . 349
9.3.4 Fonctions hyperboliques . . . 349
9.3.5 Fonctions de classifications . . . 349
9.3.6 Constantes . . . 350
9.4 decimal --- Decimal fixed-point and floating-point arithmetic . . . 351

9.4.1 Introduction pratique . . . 352
9.4.2 Les objets *Decimal* . . . 355
9.4.3 Objets de contexte . . . 363
9.4.4 Constantes . . . 369
9.4.5 Modes d'arrondi . . . 370
9.4.6 Signaux . . . 370
9.4.7 Floating-Point Notes . . . 372
9.4.8 Working with threads . . . 373
9.4.9 Cas pratiques . . . 374
9.4.10 FAQ *decimal* . . . 376
9.5 `fractions` --- Rational numbers . . . 380
9.6 `random` --- Generate pseudo-random numbers . . . 383
9.6.1 Fonctions de gestion d'état . . . 384
9.6.2 Fonctions pour les octets . . . 385
9.6.3 Fonctions pour les entiers . . . 385
9.6.4 Fonctions pour les séquences . . . 385
9.6.5 Discrete distributions . . . 386
9.6.6 Distributions pour les nombres réels . . . 387
9.6.7 Générateur alternatif . . . 388
9.6.8 Remarques sur la reproductibilité . . . 388
9.6.9 Exemples . . . 389
9.6.10 Cas pratiques . . . 391
9.6.11 Command-line usage . . . 392
9.6.12 Command-line example . . . 393
9.7 `statistics` --- Mathematical statistics functions . . . 393
9.7.1 Moyennes et mesures de la tendance centrale . . . 394
9.7.2 Mesures de la dispersion . . . 394
9.7.3 Statistics for relations between two inputs . . . 394
9.7.4 Détails des fonctions . . . 395
9.7.5 Exceptions . . . 404
9.7.6 Objets `NormalDist` . . . 404
9.7.7 Examples and Recipes . . . 406

10 Modules de programmation fonctionnelle **409**
10.1 `itertools` --- Functions creating iterators for efficient looping . . . 409
10.1.1 Itertool Functions . . . 411
10.1.2 Recettes *itertools* . . . 421
10.2 `functools` --- Higher-order functions and operations on callable objects . . . 426
10.2.1 Objets `partial` . . . 437
10.3 `operator` — Opérateurs standards en tant que fonctions . . . 437
10.3.1 Correspondances entre opérateurs et fonctions . . . 442
10.3.2 Opérateurs en-place . . . 442

11 Accès aux Fichiers et aux Dossiers **445**
11.1 `pathlib` — Chemins de système de fichiers orientés objet . . . 445
11.1.1 Utilisation basique . . . 446
11.1.2 Exceptions . . . 447
11.1.3 Chemins purs . . . 447
11.1.4 Chemins concrets . . . 457
11.1.5 Pattern language . . . 469
11.1.6 Comparaison avec les outils du module `glob` . . . 470
11.1.7 Comparaison aux modules `os` et `os.path` . . . 470
11.1.8 Protocols . . . 472
11.2 `os.path` --- Common pathname manipulations . . . 472
11.3 `stat` --- Interpreting `stat()` results . . . 479
11.4 `filecmp` --- File and Directory Comparisons . . . 485
11.4.1 La classe `dircmp` . . . 485
11.5 `tempfile` --- Generate temporary files and directories . . . 487

11.5.1 Exemples . . . 491
11.5.2 Fonctions et variables obsolètes . . . 492
11.6 glob --- Unix style pathname pattern expansion . . . 492
11.6.1 Exemples . . . 494
11.7 fnmatch --- Unix filename pattern matching . . . 495
11.8 linecache --- Random access to text lines . . . 496
11.9 shutil --- High-level file operations . . . 497
11.9.1 Opérations sur les répertoires et les fichiers . . . 497
11.9.2 Archiving operations . . . 503
11.9.3 Querying the size of the output terminal . . . 506

12 Persistance des données **509**
12.1 pickle --- Python object serialization . . . 509
12.1.1 Relations aux autres modules Python . . . 509
12.1.2 Format du flux de données . . . 510
12.1.3 Interface du module . . . 511
12.1.4 Quels objets sont sérialisables ? . . . 514
12.1.5 Sérialisation des instances d'une classe . . . 515
12.1.6 Réduction personnalisée pour les types, fonctions et autres objets . . . 521
12.1.7 Tampons hors-bande . . . 522
12.1.8 Restriction des noms dans l'espace de nommage global . . . 523
12.1.9 Performances . . . 524
12.1.10 Exemples . . . 524
12.2 copyreg --- Register pickle support functions . . . 525
12.2.1 Exemple . . . 526
12.3 shelve --- Python object persistence . . . 526
12.3.1 Limites . . . 527
12.3.2 Exemple . . . 528
12.4 marshal --- Internal Python object serialization . . . 529
12.5 dbm --- Interfaces to Unix "databases" . . . 531
12.5.1 dbm.sqlite3 --- SQLite backend for dbm . . . 532
12.5.2 dbm.gnu --- GNU database manager . . . 533
12.5.3 dbm.ndbm --- New Database Manager . . . 534
12.5.4 dbm.dumb --- Portable DBM implementation . . . 535
12.6 sqlite3 --- DB-API 2.0 interface for SQLite databases . . . 536
12.6.1 Tutoriel . . . 536
12.6.2 Références . . . 539
12.6.3 How-to guides . . . 558
12.6.4 Explanation . . . 565

13 Compression de donnée et archivage **567**
13.1 zlib --- Compression compatible with **gzip** . . . 567
13.2 gzip --- Support for **gzip** files . . . 571
13.2.1 Exemples d'utilisation . . . 573
13.2.2 Interface en ligne de commande . . . 574
13.3 bz2 --- Support for **bzip2** compression . . . 574
13.3.1 (Dé)compression de fichiers . . . 574
13.3.2 (Dé)compression incrémentielle . . . 576
13.3.3 (Dé)compression en une fois . . . 577
13.3.4 Exemples d'utilisation . . . 578
13.4 lzma --- Compression using the LZMA algorithm . . . 579
13.4.1 Lire et écrire des fichiers compressés . . . 579
13.4.2 Compresser et décompresser une donnée en mémoire . . . 580
13.4.3 Divers . . . 582
13.4.4 Préciser des chaînes de filtre personnalisées . . . 583
13.4.5 Exemples . . . 583
13.5 zipfile --- Work with ZIP archives . . . 584
13.5.1 Objets ZipFile . . . 586

13.5.2 Objets *Path* . . . 590
13.5.3 Objets *PyZipFile* . . . 592
13.5.4 Objets *ZipInfo* . . . 593
13.5.5 Interface en ligne de commande . . . 594
13.5.6 Problèmes de décompression . . . 595
13.6 `tarfile` --- Read and write tar archive files . . . 595
13.6.1 Les objets *TarFile* . . . 599
13.6.2 Les objets *TarInfo* . . . 603
13.6.3 Extraction filters . . . 605
13.6.4 Interface en ligne de commande . . . 608
13.6.5 Exemples . . . 609
13.6.6 Formats *tar* pris en charge . . . 610
13.6.7 Problèmes *unicode* . . . 611

14 Formats de fichiers **613**
14.1 `csv` --- CSV File Reading and Writing . . . 613
14.1.1 Contenu du module . . . 613
14.1.2 Dialectes et paramètres de formatage . . . 617
14.1.3 Objets lecteurs . . . 618
14.1.4 Objets transcripteurs . . . 619
14.1.5 Exemples . . . 619
14.2 `configparser` --- Configuration file parser . . . 620
14.2.1 Premiers pas . . . 621
14.2.2 Types de données prises en charge . . . 623
14.2.3 Valeurs de substitution . . . 623
14.2.4 Structure des fichiers *INI* prise en change . . . 624
14.2.5 Unnamed Sections . . . 625
14.2.6 Interpolation des valeurs . . . 625
14.2.7 Protocole d'accès associatif . . . 626
14.2.8 Personnalisation du comportement de l'analyseur . . . 627
14.2.9 Legacy API Examples . . . 632
14.2.10 ConfigParser Objects . . . 633
14.2.11 RawConfigParser Objects . . . 637
14.2.12 Exceptions . . . 638
14.3 `tomllib` --- Parse TOML files . . . 639
14.3.1 Exemples . . . 640
14.3.2 Conversion Table . . . 641
14.4 `netrc` --- netrc file processing . . . 641
14.4.1 Objets *netrc* . . . 642
14.5 `plistlib` --- Generate and parse Apple `.plist` files . . . 642
14.5.1 Exemples . . . 644

15 Service de cryptographie **645**
15.1 `hashlib` --- Algorithmes de hachage sécurisés et synthèse de messages . . . 645
15.1.1 Algorithmes de hachage . . . 645
15.1.2 Utilisation . . . 646
15.1.3 Constructeurs . . . 646
15.1.4 Attributes . . . 647
15.1.5 Objets de calcul d'empreinte . . . 647
15.1.6 Empreintes de messages de taille variable SHAKE . . . 648
15.1.7 Calcul d'empreinte (ou hachage) de fichiers . . . 648
15.1.8 Dérivation de clé . . . 649
15.1.9 BLAKE2 . . . 650
15.2 `hmac` --- Keyed-Hashing for Message Authentication . . . 657
15.3 `secrets` --- Generate secure random numbers for managing secrets . . . 659
15.3.1 Nombres aléatoires . . . 659
15.3.2 Génération de jetons . . . 659
15.3.3 Autres fonctions . . . 660

15.3.4 Recettes et bonnes pratiques . . . 660

16 Services génériques du système d'exploitation **663**
16.1 `os` --- Miscellaneous operating system interfaces . . . 663
16.1.1 Noms de fichiers, arguments en ligne de commande, et variables d'environnement . . . 664
16.1.2 Le mode UTF-8 de Python . . . 664
16.1.3 Paramètres de processus . . . 665
16.1.4 Création de fichiers objets . . . 673
16.1.5 Opérations sur les descripteurs de fichiers . . . 673
16.1.6 Fichiers et répertoires . . . 686
16.1.7 Gestion des processus . . . 712
16.1.8 Interface pour l'ordonnanceur . . . 725
16.1.9 Diverses informations sur le système . . . 727
16.1.10 Nombres aléatoires . . . 729
16.2 `io` --- Core tools for working with streams . . . 730
16.2.1 Aperçu . . . 730
16.2.2 Encodage de texte . . . 731
16.2.3 Interface de haut niveau du module . . . 732
16.2.4 Class hierarchy . . . 733
16.2.5 Performances . . . 743
16.3 `time` --- Time access and conversions . . . 743
16.3.1 Fonctions . . . 744
16.3.2 Constantes d'identification d'horloge . . . 753
16.3.3 Constantes de fuseau horaire . . . 755
16.4 `logging` --- Logging facility for Python . . . 756
16.4.1 Enregistreurs . . . 757
16.4.2 Niveaux de journalisation . . . 762
16.4.3 Gestionnaires . . . 762
16.4.4 Formateurs . . . 764
16.4.5 Filtres . . . 765
16.4.6 Objets LogRecord . . . 766
16.4.7 LogRecord attributes . . . 767
16.4.8 LoggerAdapter Objects . . . 769
16.4.9 Thread Safety . . . 769
16.4.10 Fonctions de niveau module . . . 769
16.4.11 Module-Level Attributes . . . 773
16.4.12 Integration with the warnings module . . . 774
16.5 `logging.config` --- Logging configuration . . . 774
16.5.1 Configuration functions . . . 774
16.5.2 Security considerations . . . 776
16.5.3 Configuration dictionary schema . . . 776
16.5.4 Configuration file format . . . 783
16.6 `logging.handlers` --- Logging handlers . . . 786
16.6.1 Gestionnaire à flux — *StreamHandler* . . . 786
16.6.2 Gestionnaire à fichier — *FileHandler* . . . 787
16.6.3 Gestionnaire à puits sans fond — *NullHandler* . . . 787
16.6.4 Gestionnaire à fichier avec surveillance — *WatchedFileHandler* . . . 788
16.6.5 Base des gestionnaires à roulement — *BaseRotatingHandler* . . . 788
16.6.6 Gestionnaire à roulement de fichiers — *RotatingFileHandler* . . . 789
16.6.7 Gestionnaire à roulement de fichiers périodique — *TimedRotatingFileHandler* . . . 790
16.6.8 Gestionnaire à connecteur — *SocketHandler* . . . 791
16.6.9 DatagramHandler . . . 792
16.6.10 SysLogHandler . . . 793
16.6.11 NTEventLogHandler . . . 795
16.6.12 SMTPHandler . . . 796
16.6.13 MemoryHandler . . . 796
16.6.14 HTTPHandler . . . 797
16.6.15 QueueHandler . . . 798

16.6.16 QueueListener . . . 799
16.7 platform --- Access to underlying platform's identifying data . . . 800
16.7.1 Multiplateformes . . . 800
16.7.2 Plateforme Java . . . 802
16.7.3 Plateforme Windows . . . 802
16.7.4 macOS Platform . . . 803
16.7.5 iOS Platform . . . 803
16.7.6 Plateformes Unix . . . 803
16.7.7 Linux Platforms . . . 803
16.7.8 Android Platform . . . 804
16.7.9 Miscellaneous . . . 804
16.8 errno --- Standard errno system symbols . . . 804
16.9 ctypes --- A foreign function library for Python . . . 812
16.9.1 Didacticiel de *ctypes* . . . 812
16.9.2 Référence du module . . . 832

17 Command Line Interface Libraries **849**
17.1 argparse --- Parser for command-line options, arguments and subcommands . . . 849
17.1.1 Objets ArgumentParser . . . 850
17.1.2 La méthode *add_argument()* . . . 858
17.1.3 La méthode *parse_args()* . . . 868
17.1.4 Autres outils . . . 871
17.1.5 Exceptions . . . 880
17.2 optparse --- Parser for command line options . . . 894
17.2.1 Choosing an argument parsing library . . . 894
17.2.2 Introduction . . . 895
17.2.3 Background . . . 896
17.2.4 Tutoriel . . . 898
17.2.5 Reference Guide . . . 905
17.2.6 Option Callbacks . . . 914
17.2.7 Extending optparse . . . 917
17.2.8 Exceptions . . . 920
17.3 getpass --- Portable password input . . . 920
17.4 fileinput --- Iterate over lines from multiple input streams . . . 921
17.5 curses --- Terminal handling for character-cell displays . . . 923
17.5.1 Fonctions . . . 924
17.5.2 Window Objects . . . 931
17.5.3 Constantes . . . 937
17.6 curses.textpad --- Text input widget for curses programs . . . 948
17.6.1 Textbox objects . . . 948
17.7 curses.ascii --- Utilities for ASCII characters . . . 949
17.8 curses.panel --- A panel stack extension for curses . . . 953
17.8.1 Fonctions . . . 953
17.8.2 Panel Objects . . . 953

18 Exécution concourante **955**
18.1 threading --- Thread-based parallelism . . . 955
18.1.1 Données locales au fil d'exécution . . . 958
18.1.2 Objets *Threads* . . . 958
18.1.3 Verrous . . . 961
18.1.4 RLock Objects . . . 962
18.1.5 Condition Objects . . . 963
18.1.6 Semaphore Objects . . . 965
18.1.7 Event Objects . . . 966
18.1.8 Timer Objects . . . 967
18.1.9 Barrier Objects . . . 967
18.1.10 Using locks, conditions, and semaphores in the with statement . . . 969
18.2 multiprocessing --- Process-based parallelism . . . 969

18.2.1 Introduction . . . 969
18.2.2 Référence . . . 976
18.2.3 Lignes directrices de programmation . . . 1005
18.2.4 Exemples . . . 1008
18.3 `multiprocessing.shared_memory` --- Shared memory for direct access across processes . . . 1014
18.4 The `concurrent` package . . . 1020
18.5 `concurrent.futures` --- Launching parallel tasks . . . 1020
18.5.1 Executor Objects . . . 1020
18.5.2 ThreadPoolExecutor . . . 1021
18.5.3 InterpreterPoolExecutor . . . 1023
18.5.4 ProcessPoolExecutor . . . 1024
18.5.5 Future Objects . . . 1025
18.5.6 Module Functions . . . 1027
18.5.7 Exception classes . . . 1027
18.6 `subprocess` --- Subprocess management . . . 1028
18.6.1 Utiliser le module `subprocess` . . . 1028
18.6.2 Considérations de sécurité . . . 1037
18.6.3 Objets *Popen* . . . 1037
18.6.4 Utilitaires *Popen* pour Windows . . . 1039
18.6.5 Ancienne interface (*API*) haut-niveau . . . 1042
18.6.6 Remplacer les fonctions plus anciennes par le module `subprocess` . . . 1044
18.6.7 Remplacement des fonctions originales d'invocation du *shell* . . . 1046
18.6.8 Notes . . . 1046
18.7 `sched` --- Event scheduler . . . 1047
18.7.1 Scheduler Objects . . . 1048
18.8 `queue` --- A synchronized queue class . . . 1049
18.8.1 Objets `Queue` . . . 1050
18.8.2 Objets `SimpleQueue` . . . 1052
18.9 `contextvars` --- Context Variables . . . 1052
18.9.1 Variables de contexte . . . 1053
18.9.2 Gestion de contexte manuelle . . . 1054
18.9.3 Gestion avec *asyncio* . . . 1056
18.10 `_thread` --- Low-level threading API . . . 1056

19 Réseau et communication entre processus **1061**
19.1 `asyncio` --- Asynchronous I/O . . . 1061
19.1.1 Exécuteurs (*runners*) . . . 1062
19.1.2 Coroutines et tâches . . . 1064
19.1.3 Flux (*streams*) . . . 1084
19.1.4 Primitives de synchronisation . . . 1092
19.1.5 Sous-processus . . . 1097
19.1.6 Files d'attente (*queues*) . . . 1102
19.1.7 Exceptions . . . 1105
19.1.8 Call Graph Introspection . . . 1106
19.1.9 Boucle d'évènements . . . 1107
19.1.10 Futures . . . 1131
19.1.11 Transports et Protocoles . . . 1134
19.1.12 Politiques . . . 1148
19.1.13 Prise en charge de la plate-forme . . . 1150
19.1.14 Extension . . . 1151
19.1.15 Index de l'API de haut niveau . . . 1152
19.1.16 Index de l'API de bas niveau . . . 1154
19.1.17 Programmer avec *asyncio* . . . 1159
19.2 `socket` --- Low-level networking interface . . . 1162
19.2.1 Socket families . . . 1163
19.2.2 Module contents . . . 1165
19.2.3 Socket Objects . . . 1178
19.2.4 Notes on socket timeouts . . . 1185

19.2.5 Exemple . . . 1186
19.3 ssl --- TLS/SSL wrapper for socket objects . . . 1189
19.3.1 Fonctions, constantes et exceptions . . . 1190
19.3.2 SSL Sockets . . . 1202
19.3.3 SSL Contexts . . . 1206
19.3.4 Certificates . . . 1215
19.3.5 Exemples . . . 1217
19.3.6 Notes on non-blocking sockets . . . 1219
19.3.7 Memory BIO Support . . . 1220
19.3.8 SSL session . . . 1222
19.3.9 Security considerations . . . 1222
19.3.10 TLS 1.3 . . . 1223
19.4 select --- Waiting for I/O completion . . . 1224
19.4.1 /dev/poll Polling Objects . . . 1226
19.4.2 Edge and Level Trigger Polling (epoll) Objects . . . 1227
19.4.3 Polling Objects . . . 1228
19.4.4 Kqueue Objects . . . 1229
19.4.5 Kevent Objects . . . 1229
19.5 selectors --- High-level I/O multiplexing . . . 1231
19.5.1 Introduction . . . 1231
19.5.2 Classes . . . 1231
19.5.3 Exemples . . . 1234
19.6 signal --- Set handlers for asynchronous events . . . 1234
19.6.1 General rules . . . 1234
19.6.2 Module contents . . . 1235
19.6.3 Examples . . . 1241
19.6.4 Note on SIGPIPE . . . 1242
19.6.5 Note on Signal Handlers and Exceptions . . . 1242
19.7 mmap --- Memory-mapped file support . . . 1243
19.7.1 MADV_* Constants . . . 1247
19.7.2 MAP_* Constants . . . 1248

20 Traitement des données provenant d'Internet 1249
20.1 email --- An email and MIME handling package . . . 1249
20.1.1 email.message : Representing an email message . . . 1250
20.1.2 email.parser : Parsing email messages . . . 1258
20.1.3 email.generator : Generating MIME documents . . . 1262
20.1.4 email.policy : Policy Objects . . . 1264
20.1.5 email.errors : Exception and Defect classes . . . 1272
20.1.6 email.headerregistry : Custom Header Objects . . . 1273
20.1.7 email.contentmanager : Managing MIME Content . . . 1279
20.1.8 email : Exemples . . . 1281
20.1.9 email.message.Message : représentation d'un message électronique à l'aide de l'API compat32 . . . 1288
20.1.10 email.mime : Creating email and MIME objects from scratch . . . 1296
20.1.11 email.header : Internationalized headers . . . 1298
20.1.12 email.charset : Representing character sets . . . 1301
20.1.13 email.encoders : Encoders . . . 1303
20.1.14 email.utils : Miscellaneous utilities . . . 1304
20.1.15 email.iterators : Iterators . . . 1306
20.2 json --- JSON encoder and decoder . . . 1307
20.2.1 Utilisation de base . . . 1310
20.2.2 Encodeurs et décodeurs . . . 1312
20.2.3 Exceptions . . . 1314
20.2.4 Conformité au standard et Interopérabilité . . . 1314
20.2.5 Command-line interface . . . 1316
20.3 mailbox --- Manipulate mailboxes in various formats . . . 1317
20.3.1 Mailbox objects . . . 1317

20.3.2 `Message` objects . . . 1326
20.3.3 Exceptions . . . 1334
20.3.4 Exemples . . . 1335
20.4 `mimetypes` --- Map filenames to MIME types . . . 1336
20.4.1 MimeTypes Objects . . . 1338
20.5 `base64` --- Base16, Base32, Base64, Base85 Data Encodings . . . 1339
20.5.1 Considérations de sécurité . . . 1342
20.6 `binascii` --- Convert between binary and ASCII . . . 1343
20.7 `quopri` --- Encode and decode MIME quoted-printable data . . . 1345

21 Outils de traitement de balises structurées 1347
21.1 `html` --- HyperText Markup Language support . . . 1347
21.2 `html.parser` --- Simple HTML and XHTML parser . . . 1347
21.2.1 Exemple d'application de l'analyseur HTML . . . 1348
21.2.2 Méthodes de la classe `HTMLParser` . . . 1349
21.2.3 Exemples . . . 1350
21.3 `html.entities` --- Definitions of HTML general entities . . . 1352
21.4 Modules de traitement XML . . . 1353
21.4.1 Vulnérabilités XML . . . 1353
21.4.2 The `defusedxml` Package . . . 1354
21.5 `xml.etree.ElementTree` --- The ElementTree XML API . . . 1354
21.5.1 Tutoriel . . . 1355
21.5.2 Prise en charge de *XPath* . . . 1360
21.5.3 Référence . . . 1361
21.5.4 Prise en charge de *XInclude* . . . 1365
21.5.5 Référence . . . 1366
21.6 `xml.dom` --- The Document Object Model API . . . 1374
21.6.1 Contenu du module . . . 1375
21.6.2 Objets dans le DOM . . . 1375
21.6.3 Conformité . . . 1383
21.7 `xml.dom.minidom` --- Minimal DOM implementation . . . 1384
21.7.1 Objets DOM . . . 1386
21.7.2 Exemple DOM . . . 1387
21.7.3 *minidom* et le standard DOM . . . 1388
21.8 `xml.dom.pulldom` --- Support for building partial DOM trees . . . 1389
21.8.1 Objets *DOMEventStream* . . . 1390
21.9 `xml.sax` --- Support for SAX2 parsers . . . 1391
21.9.1 Les objets *SAXException* . . . 1392
21.10 `xml.sax.handler` --- Base classes for SAX handlers . . . 1392
21.10.1 Objets *ContentHandler* . . . 1394
21.10.2 Objets *DTDHandler* . . . 1397
21.10.3 Objets *EntityResolver* . . . 1397
21.10.4 Objets *ErrorHandler* . . . 1397
21.10.5 Objets *LexicalHandler* . . . 1397
21.11 `xml.sax.saxutils` --- SAX Utilities . . . 1398
21.12 `xml.sax.xmlreader` --- Interface for XML parsers . . . 1399
21.12.1 XMLReader Objects . . . 1400
21.12.2 IncrementalParser Objects . . . 1401
21.12.3 Locator Objects . . . 1401
21.12.4 InputSource Objects . . . 1401
21.12.5 The `Attributes` Interface . . . 1402
21.12.6 The `AttributesNS` Interface . . . 1402
21.13 `xml.parsers.expat` --- Fast XML parsing using Expat . . . 1403
21.13.1 Objets *XMLParser* . . . 1404
21.13.2 ExpatError Exceptions . . . 1408
21.13.3 Exemple . . . 1408
21.13.4 Content Model Descriptions . . . 1409
21.13.5 Expat error constants . . . 1410

22 Gestion des protocoles internet **1413**
22.1 webbrowser --- Convenient web-browser controller . . . 1413
22.1.1 Browser Controller Objects . . . 1415
22.2 wsgiref --- WSGI Utilities and Reference Implementation . . . 1416
22.2.1 wsgiref.util — outils pour les environnements WSGI . . . 1416
22.2.2 wsgiref.headers -- WSGI response header tools . . . 1418
22.2.3 wsgiref.simple_server -- a simple WSGI HTTP server . . . 1419
22.2.4 wsgiref.validate --- WSGI conformance checker . . . 1420
22.2.5 wsgiref.handlers -- server/gateway base classes . . . 1421
22.2.6 wsgiref.types -- WSGI types for static type checking . . . 1424
22.2.7 Exemples . . . 1424
22.3 urllib --- URL handling modules . . . 1426
22.4 urllib.request --- Extensible library for opening URLs . . . 1426
22.4.1 Request Objects . . . 1431
22.4.2 OpenerDirector Objects . . . 1433
22.4.3 BaseHandler Objects . . . 1434
22.4.4 HTTPRedirectHandler Objects . . . 1435
22.4.5 HTTPCookieProcessor Objects . . . 1436
22.4.6 ProxyHandler Objects . . . 1436
22.4.7 HTTPPasswordMgr Objects . . . 1436
22.4.8 HTTPPasswordMgrWithPriorAuth Objects . . . 1436
22.4.9 AbstractBasicAuthHandler Objects . . . 1436
22.4.10 HTTPBasicAuthHandler Objects . . . 1437
22.4.11 ProxyBasicAuthHandler Objects . . . 1437
22.4.12 AbstractDigestAuthHandler Objects . . . 1437
22.4.13 HTTPDigestAuthHandler Objects . . . 1437
22.4.14 ProxyDigestAuthHandler Objects . . . 1437
22.4.15 HTTPHandler Objects . . . 1437
22.4.16 HTTPSHandler Objects . . . 1437
22.4.17 FileHandler Objects . . . 1437
22.4.18 DataHandler Objects . . . 1438
22.4.19 FTPHandler Objects . . . 1438
22.4.20 CacheFTPHandler Objects . . . 1438
22.4.21 UnknownHandler Objects . . . 1438
22.4.22 HTTPErrorProcessor Objects . . . 1438
22.4.23 Exemples . . . 1438
22.4.24 Legacy interface . . . 1441
22.4.25 urllib.request Restrictions . . . 1442
22.5 urllib.response --- Response classes used by urllib . . . 1442
22.6 urllib.parse --- Parse URLs into components . . . 1443
22.6.1 URL Parsing . . . 1443
22.6.2 URL parsing security . . . 1448
22.6.3 Parsing ASCII Encoded Bytes . . . 1448
22.6.4 Structured Parse Results . . . 1449
22.6.5 URL Quoting . . . 1450
22.7 urllib.error --- Exception classes raised by urllib.request . . . 1452
22.8 urllib.robotparser --- Parser for robots.txt . . . 1452
22.9 http --- HTTP modules . . . 1454
22.9.1 Codes d'état HTTP . . . 1454
22.9.2 HTTP status category . . . 1456
22.9.3 HTTP methods . . . 1457
22.10 http.client --- HTTP protocol client . . . 1457
22.10.1 Les objets HTTPConnection . . . 1459
22.10.2 Les objets HTTPResponse . . . 1462
22.10.3 Exemples . . . 1463
22.10.4 Les objets HTTPMessage . . . 1465
22.11 ftplib --- FTP protocol client . . . 1465
22.11.1 Référence . . . 1465

22.12 poplib --- POP3 protocol client . . . 1471
22.12.1 POP3 Objects . . . 1472
22.12.2 POP3 Example . . . 1474
22.13 imaplib --- IMAP4 protocol client . . . 1474
22.13.1 IMAP4 Objects . . . 1476
22.13.2 IMAP4 Example . . . 1481
22.14 smtplib --- SMTP protocol client . . . 1482
22.14.1 SMTP Objects . . . 1484
22.14.2 SMTP Example . . . 1487
22.15 uuid --- UUID objects according to **RFC 9562** . . . 1488
22.15.1 Utilisation en ligne de commande. . . . 1492
22.15.2 Exemple . . . 1492
22.15.3 Command-Line Example . . . 1493
22.16 socketserver --- A framework for network servers . . . 1493
22.16.1 Notes sur la création de serveurs . . . 1494
22.16.2 Objets serveur . . . 1496
22.16.3 Objets gestionnaire de requêtes . . . 1498
22.16.4 Exemples . . . 1499
22.17 http.server --- HTTP servers . . . 1502
22.17.1 Considérations de sécurité . . . 1508
22.18 http.cookies --- HTTP state management . . . 1509
22.18.1 Objets *Cookie* . . . 1509
22.18.2 Objets *Morsel* . . . 1510
22.18.3 Exemple . . . 1511
22.19 http.cookiejar --- Cookie handling for HTTP clients . . . 1512
22.19.1 CookieJar and FileCookieJar Objects . . . 1514
22.19.2 FileCookieJar subclasses and co-operation with web browsers . . . 1516
22.19.3 CookiePolicy Objects . . . 1516
22.19.4 DefaultCookiePolicy Objects . . . 1517
22.19.5 Objets *Cookie* . . . 1519
22.19.6 Exemples . . . 1520
22.20 xmlrpc --- XMLRPC server and client modules . . . 1521
22.21 xmlrpc.client --- XML-RPC client access . . . 1521
22.21.1 ServerProxy Objects . . . 1523
22.21.2 Objets DateTime . . . 1524
22.21.3 Binary Objects . . . 1524
22.21.4 Fault Objects . . . 1525
22.21.5 ProtocolError Objects . . . 1526
22.21.6 MultiCall Objects . . . 1526
22.21.7 Convenience Functions . . . 1527
22.21.8 Example of Client Usage . . . 1528
22.21.9 Example of Client and Server Usage . . . 1528
22.22 xmlrpc.server --- Basic XML-RPC servers . . . 1528
22.22.1 SimpleXMLRPCServer Objects . . . 1529
22.22.2 CGIXMLRPCRequestHandler . . . 1532
22.22.3 Documenting XMLRPC server . . . 1533
22.22.4 DocXMLRPCServer Objects . . . 1533
22.22.5 DocCGIXMLRPCRequestHandler . . . 1534
22.23 ipaddress --- IPv4/IPv6 manipulation library . . . 1534
22.23.1 Fonctions fabriques pratiques . . . 1534
22.23.2 Adresses IP . . . 1535
22.23.3 Définitions de réseaux IP . . . 1540
22.23.4 Objets interface . . . 1545
22.23.5 Autres fonctions au niveau de module . . . 1546
22.23.6 Exceptions personnalisées . . . 1547

23 Services multimédia **1549**
23.1 wave --- Read and write WAV files . . . 1549

23.1.1 Objets Wave_read . . . 1550
23.1.2 Objets Wave_write . . . 1551
23.2 colorsys --- Conversions between color systems . . . 1552

24 Internationalisation **1553**
24.1 gettext --- Multilingual internationalization services . . . 1553
24.1.1 API GNU **gettext** . . . 1553
24.1.2 API basée sur les classes . . . 1554
24.1.3 Internationaliser vos programmes et modules . . . 1558
24.1.4 Remerciements . . . 1560
24.2 locale --- Internationalization services . . . 1561
24.2.1 Contexte, détails, conseils, astuces et mises en garde . . . 1568
24.2.2 Pour les auteurs d'extensions et les programmes qui intègrent Python . . . 1568
24.2.3 Accéder aux catalogues de messages . . . 1569

25 Cadriciels d'applications **1571**
25.1 turtle — Tortue graphique . . . 1571
25.1.1 Introduction . . . 1571
25.1.2 Get started . . . 1571
25.1.3 Tutoriel . . . 1572
25.1.4 How to... . . . 1574
25.1.5 Turtle graphics reference . . . 1575
25.1.6 Méthodes de *RawTurtle*/*Turtle* et leurs fonctions correspondantes . . . 1578
25.1.7 Méthodes de TurtleScreen/Screen et leurs fonctions correspondantes . . . 1595
25.1.8 Classes publiques . . . 1602
25.1.9 Explication . . . 1603
25.1.10 Aide et configuration . . . 1604
25.1.11 turtledemo — Scripts de démonstration . . . 1606
25.1.12 Modifications depuis Python 2.6 . . . 1607
25.1.13 Modifications depuis Python 3.0 . . . 1608
25.2 cmd --- Support for line-oriented command interpreters . . . 1608
25.2.1 Objets Cmd . . . 1608
25.2.2 Exemple . . . 1610
25.3 shlex --- Simple lexical analysis . . . 1613
25.3.1 shlex Objects . . . 1615
25.3.2 Parsing Rules . . . 1617
25.3.3 Improved Compatibility with Shells . . . 1617

26 Interfaces Utilisateur Graphiques avec Tk **1619**
26.1 tkinter --- Python interface to Tcl/Tk . . . 1619
26.1.1 Architecture . . . 1620
26.1.2 Modules *Tkinter* . . . 1621
26.1.3 Guide de survie *Tkinter* . . . 1622
26.1.4 Fils d'exécution multiples . . . 1625
26.1.5 Guide pratique . . . 1626
26.1.6 Gestionnaires de fichiers . . . 1632
26.2 tkinter.colorchooser --- Color choosing dialog . . . 1632
26.3 tkinter.font --- Tkinter font wrapper . . . 1633
26.4 Boîtes de dialogue *Tkinter* . . . 1634
26.4.1 tkinter.simpledialog – Boîtes de dialogue de saisie standard de *Tkinter* . . . 1634
26.4.2 tkinter.filedialog – Boîtes de dialogue de sélection de fichiers . . . 1634
26.4.3 tkinter.commondialog – Modèles de fenêtre de dialogue . . . 1637
26.5 tkinter.messagebox --- Tkinter message prompts . . . 1637
26.6 tkinter.scrolledtext --- Scrolled Text Widget . . . 1639
26.7 tkinter.dnd --- Drag and drop support . . . 1639
26.8 tkinter.ttk --- Tk themed widgets . . . 1640
26.8.1 Utilisation de *Ttk* . . . 1641
26.8.2 Widgets *Ttk* . . . 1641
26.8.3 Widget . . . 1641

26.8.4 Combobox . . . 1644
26.8.5 Spinbox . . . 1645
26.8.6 Carnet de notes (*notebook*) . . . 1646
26.8.7 Barre de progression . . . 1648
26.8.8 Séparateur . . . 1648
26.8.9 Poignée de redimensionnement . . . 1649
26.8.10 Arborescence . . . 1649
26.8.11 Style Ttk . . . 1654
26.9 IDLE --- Python editor and shell . . . 1659
26.9.1 Menus . . . 1659
26.9.2 Editing and Navigation . . . 1664
26.9.3 Startup and Code Execution . . . 1666
26.9.4 Help and Preferences . . . 1669
26.9.5 idlelib --- implementation of IDLE application . . . 1670

27 Outils de développement **1671**
27.1 `typing` — Prise en charge des annotations de type . . . 1671
27.1.1 Specification for the Python Type System . . . 1672
27.1.2 Alias de type . . . 1672
27.1.3 *NewType* . . . 1673
27.1.4 Annotating callable objects . . . 1674
27.1.5 Génériques . . . 1675
27.1.6 Annotating tuples . . . 1676
27.1.7 The type of class objects . . . 1677
27.1.8 Annotating generators and coroutines . . . 1677
27.1.9 Types génériques définis par l'utilisateur . . . 1679
27.1.10 Le type `Any` . . . 1681
27.1.11 Sous-typage nominal et sous-typage structurel . . . 1683
27.1.12 Classe de données . . . 1683
27.1.13 Étapes d'Obsolescence des Fonctionnalités Majeures . . . 1724
27.2 `pydoc` --- Documentation generator and online help system . . . 1724
27.3 Python Development Mode . . . 1725
27.3.1 Effects of the Python Development Mode . . . 1725
27.3.2 ResourceWarning Example . . . 1726
27.3.3 Bad file descriptor error example . . . 1727
27.4 `doctest` — Exemples de tests interactifs en Python . . . 1728
27.4.1 Utilisation simple : vérifier des exemples dans des *docstrings* . . . 1730
27.4.2 Utilisation simple : vérifier des exemples dans un fichier texte . . . 1731
27.4.3 Comment ça marche . . . 1732
27.4.4 API de base . . . 1739
27.4.5 API de tests unitaires . . . 1741
27.4.6 API avancé . . . 1743
27.4.7 Débogage . . . 1747
27.4.8 Éditorial . . . 1750
27.5 `unittest` --- Unit testing framework . . . 1751
27.5.1 Exemple basique . . . 1752
27.5.2 Interface en ligne de commande . . . 1753
27.5.3 Découverte des tests . . . 1755
27.5.4 Organiser le code de test . . . 1756
27.5.5 Réutilisation d'ancien code de test . . . 1758
27.5.6 Ignorer des tests et des erreurs prévisibles . . . 1758
27.5.7 Distinguer les itérations de test à l'aide de sous-tests . . . 1760
27.5.8 Classes et fonctions . . . 1761
27.5.9 Classes et modules d'aménagements des tests . . . 1781
27.5.10 Traitement des signaux . . . 1783
27.6 `unittest.mock` --- mock object library . . . 1783
27.6.1 Guide rapide . . . 1784
27.6.2 La classe *Mock* . . . 1786

27.6.3 The patchers . . . 1803
27.6.4 MagicMock and magic method support . . . 1812
27.6.5 Helpers . . . 1815
27.6.6 Order of precedence of `side_effect`, `return_value` and *wraps* . . . 1823
27.7 `unittest.mock` --- getting started . . . 1825
27.7.1 Utilisation de Mock ou l'art de singer . . . 1825
27.7.2 Patch Decorators . . . 1830
27.7.3 Further Examples . . . 1832
27.8 `test` --- Regression tests package for Python . . . 1844
27.8.1 Writing Unit Tests for the `test` package . . . 1845
27.8.2 Running tests using the command-line interface . . . 1846
27.9 `test.support` --- Utilities for the Python test suite . . . 1847
27.10 `test.support.socket_helper` --- Utilities for socket tests . . . 1856
27.11 `test.support.script_helper` --- Utilities for the Python execution tests . . . 1857
27.12 `test.support.bytecode_helper` --- Support tools for testing correct bytecode generation . . . 1858
27.13 `test.support.threading_helper` --- Utilities for threading tests . . . 1858
27.14 `test.support.os_helper` --- Utilities for os tests . . . 1859
27.15 `test.support.import_helper` --- Utilities for import tests . . . 1861
27.16 `test.support.warnings_helper` --- Utilities for warnings tests . . . 1862

28 Débogueur et instrumentation **1865**
28.1 Table des évènements d'audit . . . 1865
28.2 `bdb` --- Debugger framework . . . 1869
28.3 `faulthandler` --- Dump the Python traceback . . . 1874
28.3.1 Dumping the traceback . . . 1874
28.3.2 Fault handler state . . . 1875
28.3.3 Dumping the tracebacks after a timeout . . . 1875
28.3.4 Dumping the traceback on a user signal . . . 1875
28.3.5 Issue with file descriptors . . . 1876
28.3.6 Exemple . . . 1876
28.4 `pdb` — Le débogueur Python . . . 1876
28.4.1 Commande du débogueur . . . 1879
28.5 The Python Profilers . . . 1885
28.5.1 Introduction to the profilers . . . 1885
28.5.2 Instant User's Manual . . . 1886
28.5.3 `profile` and `cProfile` Module Reference . . . 1888
28.5.4 The `Stats` Class . . . 1889
28.5.5 What Is Deterministic Profiling ? . . . 1892
28.5.6 Limitations . . . 1892
28.5.7 Calibration . . . 1892
28.5.8 Using a custom timer . . . 1893
28.6 `timeit` --- Measure execution time of small code snippets . . . 1893
28.6.1 Exemples simples . . . 1894
28.6.2 Interface Python . . . 1894
28.6.3 Interface en ligne de commande . . . 1896
28.6.4 Exemples . . . 1897
28.7 `trace` --- Trace or track Python statement execution . . . 1898
28.7.1 Utilisation en ligne de commande. . . . 1899
28.7.2 Programmatic Interface . . . 1900
28.8 `tracemalloc` --- Trace memory allocations . . . 1901
28.8.1 Exemples . . . 1901
28.8.2 API . . . 1906

29 Paquets et distribution de paquets logiciels **1913**
29.1 `ensurepip` --- Bootstrapping the `pip` installer . . . 1913
29.1.1 Interface en ligne de commande . . . 1914
29.1.2 API du module . . . 1914
29.2 `venv` --- Creation of virtual environments . . . 1915

29.2.1 Création d'environnements virtuels . . . 1915
29.2.2 How venvs work . . . 1917
29.2.3 API . . . 1918
29.2.4 Un exemple d'extension de `EnvBuilder` . . . 1920
29.3 `zipapp` --- Manage executable Python zip archives . . . 1924
29.3.1 Exemple de base . . . 1924
29.3.2 Interface en ligne de commande . . . 1925
29.3.3 API Python . . . 1925
29.3.4 Exemples . . . 1926
29.3.5 Spécification de l'interprète . . . 1927
29.3.6 Création d'applications autonomes avec *zipapp* . . . 1927
29.3.7 Le format d'archive d'application Zip Python . . . 1928

30 Environnement d'exécution Python **1929**
30.1 `sys` --- System-specific parameters and functions . . . 1929
30.2 `sys.monitoring` --- Execution event monitoring . . . 1955
30.2.1 Tool identifiers . . . 1955
30.2.2 Events . . . 1956
30.2.3 Turning events on and off . . . 1958
30.2.4 Registering callback functions . . . 1959
30.3 `sysconfig` --- Provide access to Python's configuration information . . . 1959
30.3.1 Configuration variables . . . 1960
30.3.2 Installation paths . . . 1960
30.3.3 User scheme . . . 1961
30.3.4 Home scheme . . . 1961
30.3.5 Prefix scheme . . . 1962
30.3.6 Installation path functions . . . 1963
30.3.7 Autres fonctions . . . 1964
30.3.8 Using `sysconfig` as a script . . . 1964
30.4 `builtins` --- Built-in objects . . . 1965
30.5 `__main__` --- Top-level code environment . . . 1966
30.5.1 `__name__ == '__main__'` . . . 1966
30.5.2 Le fichier `__main__.py` dans les paquets Python . . . 1968
30.5.3 `import __main__` . . . 1969
30.6 `warnings` --- Warning control . . . 1971
30.6.1 Catégories d'avertissement . . . 1971
30.6.2 Le filtre des avertissements . . . 1972
30.6.3 Suppression temporaire des avertissements . . . 1974
30.6.4 Tester les avertissements . . . 1975
30.6.5 Mise à jour du code pour les nouvelles versions des dépendances . . . 1975
30.6.6 Fonctions disponibles . . . 1976
30.6.7 Gestionnaires de contexte disponibles . . . 1978
30.7 `dataclasses` --- Data Classes . . . 1978
30.7.1 Classe de données . . . 1979
30.7.2 Post-initialisation . . . 1985
30.7.3 Variables de classe . . . 1986
30.7.4 Variables d'initialisation . . . 1986
30.7.5 Instances figées . . . 1986
30.7.6 Héritage . . . 1986
30.7.7 Re-ordering of keyword-only parameters in `__init__()` . . . 1987
30.7.8 Fabriques de valeurs par défaut . . . 1987
30.7.9 Valeurs par défaut mutables . . . 1987
30.7.10 Descriptor-typed fields . . . 1988
30.8 `contextlib` — Utilitaires pour les contextes s'appuyant sur l'instruction `with` . . . 1989
30.8.1 Utilitaires . . . 1989
30.8.2 Exemples et Recettes . . . 1998
30.8.3 Gestionnaires de contexte à usage unique, réutilisables et réentrants . . . 2002
30.9 `abc` --- Abstract Base Classes . . . 2004

30.10 atexit --- Exit handlers . . . 2009
30.10.1 Exemple avec atexit . . . 2010
30.11 traceback --- Print or retrieve a stack traceback . . . 2010
30.11.1 Fonctions de niveau module . . . 2011
30.11.2 TracebackException Objects . . . 2013
30.11.3 StackSummary Objects . . . 2015
30.11.4 FrameSummary Objects . . . 2015
30.11.5 Examples of Using the Module-Level Functions . . . 2016
30.11.6 Examples of Using TracebackException . . . 2018
30.12 __future__ --- Future statement definitions . . . 2020
30.12.1 Contenu du module . . . 2020
30.13 gc --- Garbage Collector interface . . . 2021
30.14 inspect --- Inspect live objects . . . 2025
30.14.1 Types et membres . . . 2026
30.14.2 Récupération du code source . . . 2030
30.14.3 Introspection des appelables avec l'objet Signature . . . 2031
30.14.4 Classes et fonctions . . . 2036
30.14.5 La pile d'interpréteur . . . 2039
30.14.6 Recherche dynamique d'attributs . . . 2041
30.14.7 Current State of Generators, Coroutines, and Asynchronous Generators . . . 2042
30.14.8 Bit d'option des objets code . . . 2043
30.14.9 Buffer flags . . . 2044
30.14.10 Interface en ligne de commande . . . 2044
30.15 annotationlib --- Functionality for introspecting annotations . . . 2045
30.15.1 Annotation semantics . . . 2045
30.15.2 Classes . . . 2046
30.15.3 Functions . . . 2047
30.16 site --- Site-specific configuration hook . . . 2049
30.16.1 sitecustomize . . . 2050
30.16.2 usercustomize . . . 2050
30.16.3 Readline configuration . . . 2051
30.16.4 Module contents . . . 2051
30.16.5 Interface en ligne de commande . . . 2052

31 Interpréteurs Python personnalisés 2053
31.1 code --- Interpreter base classes . . . 2053
31.1.1 Interactive Interpreter Objects . . . 2054
31.1.2 Interactive Console Objects . . . 2054
31.2 codeop --- Compile Python code . . . 2055

32 Importer des modules 2057
32.1 zipimport --- Import modules from Zip archives . . . 2057
32.1.1 Objets *zimporter* . . . 2058
32.1.2 Exemples . . . 2059
32.2 pkgutil --- Package extension utility . . . 2059
32.3 modulefinder --- Find modules used by a script . . . 2062
32.3.1 Exemples d'utilisation de la classe ModuleFinder . . . 2062
32.4 runpy --- Locating and executing Python modules . . . 2063
32.5 importlib --- The implementation of import . . . 2066
32.5.1 Introduction . . . 2066
32.5.2 Fonctions . . . 2067
32.5.3 importlib.abc -- Abstract base classes related to import . . . 2068
32.5.4 importlib.machinery -- Importers and path hooks . . . 2074
32.5.5 importlib.util -- Utility code for importers . . . 2079
32.5.6 Exemples . . . 2082
32.6 importlib.resources -- Package resource reading, opening and access . . . 2085
32.6.1 API par fonction . . . 2086
32.7 importlib.resources.abc -- Abstract base classes for resources . . . 2088

32.8 importlib.metadata -- Accessing package metadata . . . 2090
32.8.1 Aperçu . . . 2090
32.8.2 API par fonction . . . 2091
32.8.3 Distributions . . . 2094
32.8.4 Distribution Discovery . . . 2095
32.8.5 Implementing Custom Providers . . . 2095
32.9 The initialization of the sys.path module search path . . . 2097
32.9.1 Virtual Environments . . . 2098
32.9.2 _pth files . . . 2098
32.9.3 Embedded Python . . . 2099

33 Services du Langage Python **2101**
33.1 ast --- Abstract Syntax Trees . . . 2101
33.1.1 Grammaire abstraite . . . 2101
33.1.2 Classes de nœuds . . . 2104
33.1.3 Outils du module ast . . . 2134
33.1.4 Options du compilateur . . . 2138
33.1.5 Utilisation en ligne de commande . . . 2138
33.2 symtable --- Access to the compiler's symbol tables . . . 2139
33.2.1 Generating Symbol Tables . . . 2139
33.2.2 Examining Symbol Tables . . . 2139
33.2.3 Utilisation en ligne de commande. . . . 2143
33.3 token --- Constants used with Python parse trees . . . 2143
33.4 keyword --- Testing for Python keywords . . . 2147
33.5 tokenize --- Tokenizer for Python source . . . 2147
33.5.1 Analyse Lexicale . . . 2148
33.5.2 Utilisation en ligne de commande. . . . 2149
33.5.3 Exemples . . . 2149
33.6 tabnanny --- Detection of ambiguous indentation . . . 2151
33.7 pyclbr --- Python module browser support . . . 2152
33.7.1 Objets fonctions . . . 2153
33.7.2 Objets classes . . . 2153
33.8 py_compile --- Compile Python source files . . . 2154
33.8.1 Interface en ligne de commande . . . 2155
33.9 compileall --- Byte-compile Python libraries . . . 2156
33.9.1 Utilisation en ligne de commande . . . 2156
33.9.2 Fonctions publiques . . . 2157
33.10 dis --- Disassembler for Python bytecode . . . 2160
33.10.1 Command-line interface . . . 2160
33.10.2 Analyse du code intermédiaire . . . 2161
33.10.3 Analyse de fonctions . . . 2162
33.10.4 Les instructions du code intermédiaire en Python . . . 2164
33.10.5 Opcode collections . . . 2180
33.11 pickletools --- Tools for pickle developers . . . 2181
33.11.1 Utilisation de la ligne de commande . . . 2181
33.11.2 Programmatic Interface . . . 2182

34 Services spécifiques à MS Windows **2183**
34.1 msvcrt --- Useful routines from the MS VC++ runtime . . . 2183
34.1.1 Opérations sur les fichiers . . . 2183
34.1.2 Entrées-sorties sur un terminal . . . 2184
34.1.3 Autres fonctions . . . 2184
34.2 winreg --- Windows registry access . . . 2186
34.2.1 Fonctions . . . 2186
34.2.2 Constantes . . . 2191
34.2.3 Registry Handle Objects . . . 2193
34.3 winsound --- Sound-playing interface for Windows . . . 2194

35 Services spécifiques à Unix **2197**
35.1 posix --- The most common POSIX system calls . . . 2197
35.1.1 Prise en charge de gros fichiers . . . 2197
35.1.2 Contenu du Module . . . 2198
35.2 pwd --- The password database . . . 2198
35.3 grp --- The group database . . . 2199
35.4 termios --- POSIX style tty control . . . 2200
35.4.1 Exemple . . . 2201
35.5 tty --- Terminal control functions . . . 2201
35.6 pty --- Pseudo-terminal utilities . . . 2202
35.6.1 Exemple . . . 2203
35.7 fcntl --- The fcntl and ioctl system calls . . . 2204
35.8 resource --- Resource usage information . . . 2206
35.8.1 Resource Limits . . . 2207
35.8.2 Resource Usage . . . 2209
35.9 syslog --- Unix syslog library routines . . . 2210
35.9.1 Exemples . . . 2212

36 Modules command-line interface (CLI) **2215**

37 Modules remplacés **2217**
37.1 getopt --- C-style parser for command line options . . . 2217

38 Removed Modules **2221**
38.1 aifc --- Read and write AIFF and AIFC files . . . 2221
38.2 asynchat --- Asynchronous socket command/response handler . . . 2221
38.3 asyncore --- Asynchronous socket handler . . . 2221
38.4 audioop --- Manipulate raw audio data . . . 2222
38.5 cgi --- Common Gateway Interface support . . . 2222
38.6 cgitb --- Traceback manager for CGI scripts . . . 2222
38.7 chunk --- Read IFF chunked data . . . 2222
38.8 crypt --- Function to check Unix passwords . . . 2222
38.9 distutils --- Building and installing Python modules . . . 2222
38.10 imghdr --- Determine the type of an image . . . 2223
38.11 imp --- Access the import internals . . . 2223
38.12 mailcap --- Mailcap file handling . . . 2223
38.13 msilib --- Read and write Microsoft Installer files . . . 2223
38.14 nis --- Interface to Sun's NIS (Yellow Pages) . . . 2223
38.15 nntplib --- NNTP protocol client . . . 2223
38.16 ossaudiodev --- Access to OSS-compatible audio devices . . . 2224
38.17 pipes --- Interface to shell pipelines . . . 2224
38.18 smtpd --- SMTP Server . . . 2224
38.19 sndhdr --- Determine type of sound file . . . 2224
38.20 spwd --- The shadow password database . . . 2224
38.21 sunau --- Read and write Sun AU files . . . 2224
38.22 telnetlib --- Telnet client . . . 2225
38.23 uu --- Encode and decode uuencode files . . . 2225
38.24 xdrlib --- Encode and decode XDR data . . . 2225

39 Considérations de sécurité **2227**

A Glossaire **2229**

B About this documentation **2247**
B.1 Contributors to the Python documentation . . . 2247

C Histoire et licence **2249**
C.1 Histoire du logiciel . . . 2249
C.2 Conditions générales pour accéder à, ou utiliser, Python . . . 2250

C.2.1 PYTHON SOFTWARE FOUNDATION LICENSE VERSION 2 2250
C.2.2 LICENCE D'UTILISATION BEOPEN.COM POUR PYTHON 2.0 2251
C.2.3 LICENCE D'UTILISATION CNRI POUR PYTHON 1.6.1 2251
C.2.4 LICENCE D'UTILISATION CWI POUR PYTHON 0.9.0 à 1.2 2253
C.2.5 ZERO-CLAUSE BSD LICENSE FOR CODE IN THE PYTHON DOCUMENTATION . 2253
C.3 Licences et remerciements pour les logiciels tiers 2253
C.3.1 Mersenne twister 2253
C.3.2 Interfaces de connexion (*sockets*) 2254
C.3.3 Interfaces de connexion asynchrones 2255
C.3.4 Gestion de témoin (*cookie*) 2255
C.3.5 Traçage d'exécution 2256
C.3.6 Les fonctions UUencode et UUdecode 2256
C.3.7 Appel de procédures distantes en XML (*RPC*, pour *Remote Procedure Call*) 2257
C.3.8 test_epoll 2257
C.3.9 Select kqueue 2258
C.3.10 SipHash24 2258
C.3.11 *strtod* et *dtoa* 2259
C.3.12 OpenSSL 2259
C.3.13 expat 2262
C.3.14 libffi 2263
C.3.15 zlib 2263
C.3.16 cfuhash 2264
C.3.17 libmpdec 2265
C.3.18 Ensemble de tests C14N du W3C 2265
C.3.19 mimalloc 2266
C.3.20 *asyncio* 2266
C.3.21 Global Unbounded Sequences (GUS) 2267

D Copyright **2269**

Bibliographie **2271**

Index des modules Python **2273**

Index **2277**

Alors que reference-index décrit exactement la syntaxe et la sémantique du langage Python, ce manuel de référence de la Bibliothèque décrit la bibliothèque standard distribuée avec Python. Il décrit aussi certains composants optionnels typiquement inclus dans les distributions de Python.

La bibliothèque standard de Python est très grande, elle offre un large éventail d'outils comme le montre la longueur de la table des matières ci-dessous. La bibliothèque contient des modules natifs (écrits en C) exposant les fonctionnalités du système telles que les interactions avec les fichiers qui autrement ne seraient pas accessibles aux développeurs Python, ainsi que des modules écrits en Python exposant des solutions standardisées à de nombreux problèmes du quotidien du développeur. Certains de ces modules sont définis explicitement pour encourager et améliorer la portabilité des programmes Python en abstrayant des spécificités sous-jacentes en API neutres.

Les installateurs de Python pour Windows incluent généralement la bibliothèque standard en entier, et y ajoutent souvent d'autres composants. Pour les systèmes d'exploitation Unix, Python est typiquement fourni sous forme d'une collection de paquets, il peut donc être nécessaire d'utiliser le gestionnaire de paquets fourni par le système d'exploitation pour obtenir certains composants optionnels.

Au delà de la bibliothèque standard, il existe une collection grandissante de plusieurs centaine de milliers de composants (des programmes, des modules, ou des *frameworks*), disponibles dans le Python Package Index.

CHAPITRE 1

Introduction

La « Bibliothèque Python » contient divers composants dans différentes catégories.

Elle contient des types de données qui seraient normalement considérés comme « fondamentaux » au langage, tel que les nombres et les listes. Pour ces types, le cœur du langage en définit les écritures littérales et impose quelques contraintes sémantiques, sans les définir exhaustivement. (Cependant le cœur du langage impose quelques propriétés comme l'orthographe des attributs ou les caractéristiques des opérateurs.)

La bibliothèque contient aussi des fonctions et des exceptions natives, pouvant être utilisées par tout code Python sans `import`. Certaines sont définies par le noyau de Python, bien qu'elles ne soient pas toutes essentielles.

La grande majorité de la bibliothèque consiste cependant en une collection de modules. Cette collection peut être parcourue de différentes manières. Certains modules sont rédigés en C et inclus dans l'interpréteur Python, d'autres sont écrits en Python et leur source est importée. Certains modules fournissent des interfaces extrêmement spécifiques à Python, tel que l'affichage d'une pile d'appels, d'autres fournissent des interfaces spécifiques à un système d'exploitation, comme l'accès à du matériel spécifique. D'autres fournissent des interfaces spécifiques à un domaine d'application, comme le *World Wide Web*. Certains modules sont disponibles dans toutes les versions et implémentations de Python, d'autres ne sont disponibles que si le système sous-jacent les gère ou en a besoin. Enfin, d'autres ne sont disponibles que si Python à été compilé avec une certaine option.

Cette documentation organise les modules « de l'intérieur vers l'extérieur », documentant en premier les fonctions natives, les types de données et exceptions, puis les modules, groupés par chapitre, par thématiques.

Ça signifie que si vous commencez à lire cette documentation du début, et sautez au chapitre suivant lorsqu'elle vous ennuie, vous aurez un aperçu global des modules et domaines couverts par cette bibliothèque. Bien sûr vous n'avez pas à la lire comme un roman, vous pouvez simplement survoler la table des matières (au début), ou chercher une fonction, un module, ou un mot dans l'index (à la fin). Et si vous appréciez apprendre sur des sujets au hasard, choisissez une page au hasard (avec le module `random`) et lisez un chapitre ou deux. Peu importe l'ordre que vous adopterez, commencez par le chapitre *Fonctions natives*, car les autres chapitres présument que vous en avez une bonne connaissance.

Que le spectacle commence !

1.1 Notes sur la disponibilité

— Une note « Disponibilité : Unix » signifie que cette fonction est communément implémentée dans les systèmes Unix. Une telle note ne prétend pas l'existence de la fonction sur un système d'exploitation particulier.
— If not separately noted, all functions that claim "Availability : Unix" are supported on macOS, iOS and Android, all of which build on a Unix core.

— Si une note de disponibilité contient à la fois une version minimale du noyau et une version minimale de la *libc*, les deux conditions doivent être remplies. Par exemple, une fonctionnalité avec la note *Disponibilité : Linux >= 3.17 avec glibc >= 2.27* nécessite à la fois Linux 3.17 ou plus récent et glibc 2.27 ou plus récent.

1.1.1 Plateformes WebAssembly

Les plates-formes WebAssembly `wasm32-emscripten` (Emscripten) et `wasm32-wasi` (WASI) fournissent un sous-ensemble des API POSIX. Les environnements d'exécution WebAssembly et les navigateurs sont dans des bacs à sable et ont un accès limité à l'hôte et aux ressources externes. Tout module de bibliothèque standard Python qui utilise des processus, des threads, le réseau, des signaux ou d'autres formes de communication inter-processus (IPC) n'est pas disponible ou peut ne pas fonctionner comme sur d'autres systèmes de type Unix. Les entrées-sorties de fichiers, le système de fichiers et les fonctions liées aux autorisations Unix sont également restreintes. *Emscripten* ne permet pas d'effectuer des entrées-sorties bloquantes. D'autres opérations bloquantes comme `sleep()` bloquent la boucle d'événements du navigateur.

Les propriétés et le comportement de Python sur les plates-formes WebAssembly dépendent de la version du SDK Emscripten ou WASI, des programmes exécutables WASM (navigateur, NodeJS, wasmtime) et de la configuration de compilation de Python. WebAssembly, Emscripten et WASI sont des normes en évolution ; certaines fonctionnalités telles que la gestion du réseau pourraient être prises en charge à l'avenir.

Pour exécuter du Python dans le navigateur, les utilisateurs doivent envisager Pyodide ou PyScript. *PyScript* est construit sur *Pyodide*, qui est lui-même construit sur CPython et *Emscripten*. *Pyodide* donne accès aux API JavaScript et DOM des navigateurs ainsi qu'à des capacités réseau limitées avec les API `XMLHttpRequest` et `Fetch` de JavaScript.

— Les API liées aux processus ne sont pas disponibles ou échouent toujours avec une erreur. Cela inclut les API qui génèrent de nouveaux processus (`fork()`, `execve()`), attendent des processus (`waitpid()`), envoient des signaux (`kill()`), ou interagissent d'une autre manière avec les processus. Le module `subprocess` peut être importé mais ne fonctionne pas.
— Le module `socket` est disponible, mais est limité et n'a pas le même comportement que sur les autres plates-formes. Sur *Emscripten*, les connecteurs réseau sont toujours non bloquants et nécessitent du code JavaScript et du code supplémentaire sur le serveur pour faire mandataire TCP via *WebSockets* ; voir Emscripten Networking pour plus d'informations. La pré-version 1 de *WASI* n'autorise que les connecteurs réseau utilisant un descripteur de fichier déjà existant.
— Certaines fonctions sont des émulations embryonnaires qui ne font rien et renvoient toujours des valeurs codées en dur.
— Les fonctions liées aux descripteurs de fichiers, aux autorisations de fichiers, aux propriétaires des fichiers et aux liens sont limitées et ne prennent pas en charge certaines opérations. Par exemple, WASI n'autorise pas les liens symboliques avec des noms de fichiers absolus.

1.1.2 Mobile platforms

Android and iOS are, in most respects, POSIX operating systems. File I/O, socket handling, and threading all behave as they would on any POSIX operating system. However, there are several major differences :

— Mobile platforms can only use Python in "embedded" mode. There is no Python REPL, and no ability to use separate executables such as **`python`** or **`pip`**. To add Python code to your mobile app, you must use the Python embedding API. For more details, see using-android and using-ios.
— Subprocesses :
  — On Android, creating subprocesses is possible but officially unsupported. In particular, Android does not support any part of the System V IPC API, so `multiprocessing` is not available.
  — An iOS app cannot use any form of subprocessing, multiprocessing, or inter-process communication. If an iOS app attempts to create a subprocess, the process creating the subprocess will either lock up, or crash. An iOS app has no visibility of other applications that are running, nor any ability to communicate with other running applications, outside of the iOS-specific APIs that exist for this purpose.
— Mobile apps have limited access to modify system resources (such as the system clock). These resources will often be *readable*, but attempts to modify those resources will usually fail.
— Console input and output :
  — On Android, the native `stdout` and `stderr` are not connected to anything, so Python installs its own streams which redirect messages to the system log. These can be seen under the tags `python.stdout` and `python.stderr` respectively.

— iOS apps have a limited concept of console output. `stdout` and `stderr` *exist*, and content written to `stdout` and `stderr` will be visible in logs when running in Xcode, but this content *won't* be recorded in the system log. If a user who has installed your app provides their app logs as a diagnostic aid, they will not include any detail written to `stdout` or `stderr`.
— Mobile apps have no usable `stdin` at all. While apps can display an on-screen keyboard, this is a software feature, not something that is attached to `stdin`.

As a result, Python modules that involve console manipulation (such as *curses* and *readline*) are not available on mobile platforms.

CHAPITRE 2

Fonctions natives

L'interpréteur Python propose quelques fonctions et types natifs qui sont toujours disponibles. Ils sont listés ici par ordre alphabétique.

Fonctions natives

A
abs()
aiter()
all()
anext()
any()
ascii()

B
bin()
bool()
breakpoint()
bytearray()
bytes()

C
callable()
chr()
classmethod()
compile()
complex()

D
delattr()
dict()
dir()
divmod()

E
enumerate()
eval()
exec()

F
filter()
float()
format()
frozenset()

G
getattr()
globals()

H
hasattr()
hash()
help()
hex()

I
id()
input()
int()
isinstance()
issubclass()
iter()

L
len()
list()
locals()

M
map()
max()
memoryview()
min()

N
next()

O
object()
oct()
open()
ord()

P
pow()
print()
property()

R
range()
repr()
reversed()
round()

S
set()
setattr()
slice()
sorted()
staticmethod()
str()
sum()
super()

T
tuple()
type()

V
vars()

Z
zip()

_
__import__()

abs(*x*)

Renvoie la valeur absolue d'un nombre. L'argument peut être un nombre entier, un nombre à virgule flottante, ou tout objet définissant `__abs__()`. Si l'argument est un nombre complexe, son module est renvoyé.

aiter(*async_iterable*)

Renvoie un *itérateur asynchrone* pour l'*itérable asynchrone* donné. Équivaut à appeler `x.__aiter__()`.

Remarque : contrairement à `iter()`, `aiter()` n'a pas de variante à 2 arguments.

Ajouté dans la version 3.10.

all(*iterable*)

Renvoie `True` si tous les éléments de *iterable* sont vrais (ou s'il est vide). Équivaut à :

```
def all(iterable):
    for element in iterable:
        if not element:
            return False
    return True
```

awaitable anext(*async_iterator*)

awaitable anext (*async_iterator*, *default*)

Lorsqu'il est attendu, renvoie l'élément suivant à partir de l'*itérateur asynchrone* donné, ou *default* s'il est fourni et que l'itérateur est épuisé.

Il s'agit de la variante asynchrone de la fonction native `next()` et elle se comporte de la même manière.

Renvoie un *attendable* en appelant la méthode `__anext__()` de *async_iterator*. L'attente renvoie la prochaine valeur de l'itérateur. Si *default* est fourni, il est renvoyé si l'itérateur est épuisé, sinon `StopAsyncIteration` est levée.

Ajouté dans la version 3.10.

any (*iterable*)

Renvoie `True` si au moins un élément de *iterable* est vrai. `False` est renvoyé dans le cas où *iterable* est vide. Équivaut à :

```
def any(iterable):
    for element in iterable:
        if element:
            return True
    return False
```

ascii (*object*)

Renvoie, tout comme `repr()`, une chaîne contenant une représentation d'un objet destinée à l'affichage, mais en transformant les caractères non ASCII renvoyés par `repr()` par l'utilisation de séquences d'échappement `\x`, `\u` ou `\U`. Cela génère une chaîne similaire à ce que renvoie `repr()` dans Python 2.

bin (*x*)

Convertit un nombre entier en binaire dans une chaîne avec le préfixe "0b". Le résultat est une expression Python valide. Si *x* n'est pas un `int`, il doit définir une méthode `__index__()` donnant un nombre entier. Voici quelques exemples :

```
>>> bin(3)
'0b11'
>>> bin(-10)
'-0b1010'
```

Vous pouvez contrôler l'affichage du préfixe `"0b"` à l'aide d'un des moyens suivants.

```
>>> format(14, '#b'), format(14, 'b')
('0b1110', '1110')
>>> f'{14:#b}', f'{14:b}'
('0b1110', '1110')
```

Voir aussi `format()` pour plus d'informations.

class bool (*object=False*, /)

Renvoie une valeur booléenne, c'est-à-dire soit `True`, soit `False`. L'argument est converti en utilisant la *procédure standard d'évaluation de valeur de vérité*. Si l'argument est faux, ou omis, elle renvoie `False`, sinon, elle renvoie `True`. La classe `bool` hérite de la classe `int` (voir *Types numériques — int, float, complex*). Il n'est pas possible d'en hériter. Ses seules instances sont `False` et `True` (voir *Boolean Type - bool*).

Modifié dans la version 3.7 : Le paramètre est désormais un argument exclusivement positionnel.

breakpoint (**args*, ***kws*)

Cette fonction vous place dans le débogueur lorsqu'elle est appelée. Plus précisément, elle appelle `sys.breakpointhook()`, en lui passant les arguments `args` et `kws`. Par défaut, `sys.breakpointhook()` appelle `pdb.set_trace()` qui n'attend aucun argument. Dans ce cas, c'est purement une fonction de commodité donc vous n'avez pas à importer explicitement `pdb` ou à taper plus de code pour entrer dans le débogueur. Cependant, il est possible d'affecter une autre fonction à `sys.breakpointhook()`, que `breakpoint()` appellera automatiquement, vous permettant ainsi de basculer dans le débogueur de votre choix. Si `sys.breakpointhook()` n'est pas accessible, cette fonction lève `RuntimeError`.

Par défaut, le comportement de la fonction `breakpoint()` peut être changé par la variable d'environnement `PYTHONBREAKPOINT`. Voir `sys.breakpointhook()` pour plus de détails.

Notez que ceci n'est plus garanti si la fonction `sys.breakpointhook()` a été remplacée.

Lève un *évènement d'audit* `builtins.breakpoint` avec l'argument `breakpointhook`.

Ajouté dans la version 3.7.

class bytearray(*source=b"*)

class bytearray(*source*, *encoding*)

class bytearray(*source*, *encoding*, *errors*)

Renvoie un nouveau tableau d'octets. La classe `bytearray` est une séquence mutable de nombres entiers dans l'intervalle 0 ≤ x < 256. Il possède la plupart des méthodes des séquences variables, décrites dans *Types de séquences mutables*, ainsi que la plupart des méthodes de la classe `bytes`, voir *Opérations sur les bytes et bytearray*.

Le paramètre optionnel *source* peut être utilisé pour initialiser le tableau de plusieurs façons :
- si c'est une *chaîne*, vous devez aussi donner le paramètre *encoding* pour l'encodage (et éventuellement *errors*). La fonction `bytearray()` convertit ensuite la chaîne en octets *via* la méthode `str.encode()` ;
- si c'est un *entier*, le tableau a cette taille et est initialisé d'octets *null* ;
- si c'est un objet conforme à l'interface tampon, un tampon en lecture seule de l'objet est utilisé pour initialiser le tableau ;
- si c'est un *itérable*, il doit itérer sur des nombres entier dans l'intervalle `0 <= x < 256`, qui sont utilisés pour initialiser le contenu du tableau.

Sans argument, un tableau vide est créé.

Voir *Séquences Binaires — bytes, bytearray, vue mémoire* et *Objets bytearray*.

class bytes(*source=b"*)

class bytes(*source*, *encoding*)

class bytes(*source*, *encoding*, *errors*)

Renvoie un nouvel objet *bytes*, qui est une séquence immuable de nombres entiers dans l'intervalle `0 <= x < 256`. Un `bytes` est une version immuable de `bytearray` — avec les mêmes méthodes d'accès, et le même comportement lors de l'indexation ou du découpage.

En conséquence, les arguments du constructeur sont les mêmes que pour `bytearray()`.

Les objets *bytes* peuvent aussi être créés à partir de littéraux, voir strings.

Voir aussi *Séquences Binaires — bytes, bytearray, vue mémoire*, *Objets bytes*, et *Opérations sur les bytes et bytearray*.

callable(*object*)

Renvoie `True` si l'argument *object* semble être appelable, sinon `False`. Lorsqu'elle renvoie `True`, il est toujours possible qu'un appel échoue. Cependant, lorsqu'elle renvoie `False`, il n'est jamais possible d'appeler *object*. Notez qu'une classe est toujours appelable (appeler une classe donne une nouvelle instance) ; une instance n'est appelable que si sa classe définit une méthode `__call__()`.

Ajouté dans la version 3.2 : cette fonction a d'abord été supprimée avec Python 3.0 puis elle a été remise dans Python 3.2.

chr(*i*)

Renvoie la chaîne représentant un caractère dont le code de caractère Unicode est le nombre entier *i*. Par exemple, `chr(97)` renvoie la chaîne de caractères `'a'`, tandis que `chr(8364)` renvoie `'€'`. Il s'agit de la réciproque de `ord()`.

L'intervalle valide pour cet argument est de `0` à `1114111` (`0x10FFFF` en base 16). Une exception `ValueError` est levée si *i* est en dehors de l'intervalle.

@classmethod

Transforme une méthode en méthode de classe.

Une méthode de classe reçoit implicitement la classe en premier argument, tout comme une méthode d'instance reçoit l'instance. Voici comment déclarer une méthode de classe :

```
class C:
    @classmethod
    def f(cls, arg1, arg2): ...
```

La forme `@classmethod` est un *décorateur* de fonction — consultez function pour plus de détails.

Elle peut être appelée soit sur la classe (comme `C.f()`) ou sur une instance (comme `C().f()`). L'instance est ignorée, sauf pour déterminer sa classe. Si la méthode est appelée sur une instance de classe fille, c'est la classe fille qui sera donnée en premier argument implicite.

Les méthodes de classe sont différentes des méthodes statiques du C++ ou du Java. Si ce sont elles dont vous avez besoin, regardez du côté de `staticmethod()` dans cette section. Voir aussi types.

Modifié dans la version 3.9 : les méthodes de classe peuvent encapsuler d'autres *descripteurs* comme `property()`.

Modifié dans la version 3.10 : les méthodes de classe héritent dorénavant des attributs des méthodes (`__module__`, `__name__`, `__qualname__`, `__doc__` et `__annotations__`) et ont un nouvel attribut `__wrapped__`.

Deprecated since version 3.11, removed in version 3.13 : les méthodes de classe ne peuvent plus encapsuler d'autres *descripteurs* comme `property()`.

compile(*source*, *filename*, *mode*, *flags=0*, *dont_inherit=False*, *optimize=-1*)

Compile *source* en un objet code ou objet AST. Les objets code peuvent être exécutés par `exec()` ou `eval()`. *source* peut être une chaîne, une chaîne d'octets, ou un objet AST. Consultez la documentation du module `ast` pour des informations sur la manipulation d'objets AST.

L'argument *filename* doit désigner le fichier depuis lequel le code a été lu. Donnez quelque chose de reconnaissable lorsqu'il n'a pas été lu depuis un fichier (typiquement `"<string>"`).

L'argument *mode* indique quel type de code doit être compilé : `'exec'` si *source* est une suite d'instructions, `'eval'` pour une seule expression, ou `'single'` s'il ne contient qu'une instruction interactive (dans ce dernier cas, les résultats d'expressions donnant autre chose que `None` sont affichés).

Les arguments optionnels *flags* et *dont_inherit* contrôlent quelles *options de compilation* seront activées et quelles instructions future seront autorisées. Si aucun des deux n'est présent (ou que les deux sont à 0), le code est compilé avec les mêmes paramètres que le code appelant `compile()`. Si l'argument *flags* est fourni alors que *dont_inherit* ne l'est pas (ou vaut 0), les options de compilation et les instructions *futures* utilisées sont celles définies par *flags* en plus de celles qui auraient été utilisées. Si *dont_inherit* est un entier différent de zéro, *flags* est utilisé tel quel — les *flags* (instructions futures et options de compilation) valables pour le code encadrant *compile* sont ignorés.

Les instructions *future* sont contrôlées par des bits, il est ainsi possible d'en activer plusieurs en les combinant avec un *OU* binaire. Les bits requis pour demander une certaine fonctionnalité se trouvent dans l'attribut `compiler_flag` de la classe `Feature` du module `__future__`. Les *options du compilateur* se trouvent dans le module `ast`, avec le préfixe `PyCF_`.

L'argument *optimize* indique le niveau d'optimisation du compilateur. La valeur par défaut est `-1` qui prend le niveau d'optimisation de l'interpréteur tel que reçu via l'option `-O`. Les niveaux explicites sont : `0` (pas d'optimisation, `__debug__` est `True`), `1` (les `assert` sont supprimés, `__debug__` est `False`) ou `2` (les chaines de documentation sont également supprimées).

Cette fonction lève une `SyntaxError` si la source n'est pas valide, et `ValueError` si la source contient des octets *null*.

Si vous voulez transformer du code Python en sa représentation AST, voyez `ast.parse()`.

Lève un *évènement d'audit* `compile` avec les arguments `source` et `filename`. Cet événement peut également être levé par une compilation implicite.

Note

lors de la compilation d'une chaîne de plusieurs lignes de code avec les modes `'single'` ou `'eval'`, celles-ci doivent être terminées par au moins un retour à la ligne. Cela permet de faciliter la distinction entre les instructions complètes et incomplètes dans le module *code*.

Avertissement

il est possible de faire planter l'interpréteur Python avec des chaînes suffisamment grandes ou complexes lors de la compilation d'un objet AST. Ceci est dû à limitation de la profondeur de la pile d'appels.

Modifié dans la version 3.2 : autorise l'utilisation de retours à la ligne Mac et Windows. Par ailleurs, la chaîne donnée à `'exec'` n'a plus besoin de terminer par un retour à la ligne. Ajout du paramètre *optimize*.

Modifié dans la version 3.5 : précédemment, l'exception *TypeError* était levée quand un caractère nul était rencontré dans *source*.

Ajouté dans la version 3.8 : `ast.PyCF_ALLOW_TOP_LEVEL_AWAIT` peut maintenant être passée à *flags* pour permettre une gestion haut niveau de `await`, `async for` et `async with`.

class complex(*number=0*, /)

class complex(*string*, /)

class complex(*real=0*, *imag=0*)

Convert a single string or number to a complex number, or create a complex number from real and imaginary parts.

Exemples :

```
>>> complex('+1.23')
(1.23+0j)
>>> complex('-4.5j')
-4.5j
>>> complex('-1.23+4.5j')
(-1.23+4.5j)
>>> complex('\t( -1.23+4.5J )\n')
(-1.23+4.5j)
>>> complex('-Infinity+NaNj')
(-inf+nanj)
>>> complex(1.23)
(1.23+0j)
>>> complex(imag=-4.5)
-4.5j
>>> complex(-1.23, 4.5)
(-1.23+4.5j)
```

If the argument is a string, it must contain either a real part (in the same format as for *float()*) or an imaginary part (in the same format but with a `'j'` or `'J'` suffix), or both real and imaginary parts (the sign of the imaginary part is mandatory in this case). The string can optionally be surrounded by whitespaces and the round parentheses `'('` and `')'`, which are ignored. The string must not contain whitespace between `'+'`, `'-'`, the `'j'` or `'J'` suffix, and the decimal number. For example, `complex('1+2j')` is fine, but `complex('1 + 2j')` raises *ValueError*. More precisely, the input must conform to the *complexvalue* production rule in the following grammar, after parentheses and leading and trailing whitespace characters are removed :

```
complexvalue:  floatvalue |
               floatvalue ("j" | "J") |
               floatvalue sign absfloatvalue ("j" | "J")
```

Si l'argument est un nombre, le constructeur le convertit, comme le font les classes *int* et *float*. Pour les autres objets `x`, `complex(x)` délègue à `x.__complex__()`. Si `__complex__()` n'est pas définie, elle délègue à

`__float__()`. Si `__float__()` n'est pas défini, alors elle délègue à `__index__()`.

If two arguments are provided or keyword arguments are used, each argument may be any numeric type (including complex). If both arguments are real numbers, return a complex number with the real component *real* and the imaginary component *imag*. If both arguments are complex numbers, return a complex number with the real component `real.real-imag.imag` and the imaginary component `real.imag+imag.real`. If one of arguments is a real number, only its real component is used in the above expressions.

See also `complex.from_number()` which only accepts a single numeric argument.

If all arguments are omitted, returns `0j`.

Le type complexe est décrit dans *Types numériques — int, float, complex*.

Modifié dans la version 3.6 : les chiffres peuvent être groupés avec des tirets bas comme dans les expressions littérales.

Modifié dans la version 3.8 : délègue à `__index__()` si `__complex__()` et `__float__()` ne sont pas définies.

Obsolète depuis la version 3.14 : Passing a complex number as the *real* or *imag* argument is now deprecated ; it should only be passed as a single positional argument.

delattr(*object*, *name*)

C'est une cousine de `setattr()`. Les arguments sont un objet et une chaîne. La chaîne doit être le nom de l'un des attributs de l'objet. La fonction supprime l'attribut nommé, si l'objet l'y autorise. Par exemple `delattr(x, 'foobar')` est l'équivalent de `del x.foobar`. *name* n'a pas besoin d'être un identifiant Python (voir `setattr()`).

class dict(***kwarg*)

class dict(*mapping*, ***kwarg*)

class dict(*iterable*, ***kwarg*)

Crée un nouveau dictionnaire. L'objet `dict` est la classe du dictionnaire. Voir `dict` et *Les types de correspondances — dict* pour vous documenter sur cette classe.

Pour les autres conteneurs, voir les classes natives `list`, `set`, et `tuple`, ainsi que le module `collections`.

dir()

dir(*object*)

Sans argument, elle donne la liste des noms dans l'espace de nommage local. Avec un argument, elle essaye de donner une liste d'attributs valides pour cet objet.

Si l'objet a une méthode `__dir__()`, elle est appelée et doit donner une liste d'attributs. Cela permet aux objets implémentant `__getattr__()` ou `__getattribute__()` de personnaliser ce que renvoie `dir()`.

Si l'objet ne fournit pas de méthode `__dir__()`, la fonction fait de son mieux en rassemblant les informations de l'attribut `__dict__` de l'objet, s'il est défini, et depuis son type. La liste résultante n'est pas nécessairement complète, et peut être erronée quand l'objet a une `__getattr__()` personnalisée.

Le mécanisme par défaut de `dir()` se comporte différemment avec différents types d'objets, car elle préfère donner une information pertinente plutôt qu'exhaustive :
— si l'objet est un module, la liste contiendra les noms des attributs du module ;
— si l'objet est un type ou une classe, la liste contiendra les noms de ses attributs et, récursivement, des attributs de ses parents ;
— autrement, la liste contient les noms des attributs de l'objet, le nom des attributs de la classe, et récursivement des attributs des parents de la classe.
La liste donnée est triée par ordre alphabétique, par exemple :

```
>>> import struct
>>> dir()   # show the names in the module namespace
['__builtins__', '__name__', 'struct']
>>> dir(struct)   # show the names in the struct module
['Struct', '__all__', '__builtins__', '__cached__', '__doc__', '__file__',
```

(suite sur la page suivante)

(suite de la page précédente)

```
 '__initializing__', '__loader__', '__name__', '__package__',
 '_clearcache', 'calcsize', 'error', 'pack', 'pack_into',
 'unpack', 'unpack_from']
>>> class Shape:
...     def __dir__(self):
...         return ['area', 'perimeter', 'location']
...
>>> s = Shape()
>>> dir(s)
['area', 'location', 'perimeter']
```

Note

étant donné que `dir()` est d'abord fournie pour son côté pratique en mode interactif, elle a tendance à fournir un ensemble de noms pertinents plutôt qu'un ensemble exhaustif et rigoureusement défini, son comportement peut aussi changer d'une version à l'autre. Par exemple, les attributs de méta-classes ne sont pas donnés lorsque l'argument est une classe.

divmod(*a*, *b*)

Prend deux nombres (qui ne sont pas des nombres complexes) et renvoie leur quotient et reste de leur division entière sous forme d'une paire de nombres. Avec des opérandes de types différents, les règles des opérateurs binaires s'appliquent. Pour deux entiers le résultat est le même que `(a // b, a % b)`. Pour des nombres à virgule flottante le résultat est `(q, a % b)`, où *q* est généralement `math.floor(a / b)` mais peut valoir 1 de moins. Dans tous les cas `q * b + a % b` est très proche de *a*. Si `a % b` est différent de zéro, il a le même signe que *b* et `0 <= abs(a % b) < abs(b)`.

enumerate(*iterable*, *start=0*)

Renvoie un objet énumérant. *iterable* doit être une séquence, un *itérateur*, ou tout autre objet prenant en charge l'itération. La méthode `__next__()` de l'itérateur donné par `enumerate()` renvoie un *n*-uplet contenant un compte (démarrant à *start*, 0 par défaut) et les valeurs obtenues de l'itération sur *iterable*.

```
>>> seasons = ['Spring', 'Summer', 'Fall', 'Winter']
>>> list(enumerate(seasons))
[(0, 'Spring'), (1, 'Summer'), (2, 'Fall'), (3, 'Winter')]
>>> list(enumerate(seasons, start=1))
[(1, 'Spring'), (2, 'Summer'), (3, 'Fall'), (4, 'Winter')]
```

Équivalent à :

```
def enumerate(iterable, start=0):
    n = start
    for elem in iterable:
        yield n, elem
        n += 1
```

eval(*source*, */*, *globals=None*, *locals=None*)

Paramètres

- **source** (`str` | code object) -- A Python expression.
- **globals** (`dict` | `None`) -- The global namespace (default : `None`).
- **locals** (*mapping* | `None`) -- The local namespace (default : `None`).

Renvoie

The result of the evaluated expression.

Raises

Syntax errors are reported as exceptions.

> **Avertissement**
>
> This function executes arbitrary code. Calling it with user-supplied input may lead to security vulnerabilities.

The *expression* argument is parsed and evaluated as a Python expression (technically speaking, a condition list) using the *globals* and *locals* mappings as global and local namespace. If the *globals* dictionary is present and does not contain a value for the key `__builtins__`, a reference to the dictionary of the built-in module `builtins` is inserted under that key before *expression* is parsed. That way you can control what builtins are available to the executed code by inserting your own `__builtins__` dictionary into *globals* before passing it to `eval()`. If the *locals* mapping is omitted it defaults to the *globals* dictionary. If both mappings are omitted, the expression is executed with the *globals* and *locals* in the environment where `eval()` is called. Note, *eval()* will only have access to the *nested scopes* (non-locals) in the enclosing environment if they are already referenced in the scope that is calling `eval()` (e.g. via a `nonlocal` statement).

Example :

```
>>> x = 1
>>> eval('x+1')
2
```

Cette fonction peut aussi être utilisée pour exécuter n'importe quel objet code (tels que ceux créés par `compile()`). Dans ce cas, donnez un objet code plutôt qu'une chaîne. Si l'objet code a été compilé avec l'argument *mode* à `'exec'`, `eval()` renvoie `None`.

Conseils : l'exécution dynamique d'instructions est gérée par la fonction `exec()`. Les fonctions `globals()` et `locals()` renvoient respectivement les dictionnaires globaux et locaux, qui peuvent être utiles lors de l'usage de `eval()` et `exec()`.

Si la source donnée est une chaîne, les espaces de début et de fin et les tabulations sont supprimées.

Utilisez `ast.literal_eval()` si vous avez besoin d'une fonction qui peut évaluer en toute sécurité des chaînes avec des expressions ne contenant que des valeurs littérales.

Lève un *évènement d'audit* `exec` avec l'objet de code comme argument. Les événements de compilation de code peuvent également être levés.

Modifié dans la version 3.13 : The *globals* and *locals* arguments can now be passed as keywords.

Modifié dans la version 3.13 : The semantics of the default *locals* namespace have been adjusted as described for the `locals()` builtin.

exec(*source*, /, *globals=None*, *locals=None*, *, *closure=None*)

> **Avertissement**
>
> This function executes arbitrary code. Calling it with user-supplied input may lead to security vulnerabilities.

This function supports dynamic execution of Python code. *source* must be either a string or a code object. If it is a string, the string is parsed as a suite of Python statements which is then executed (unless a syntax error occurs). [1] If it is a code object, it is simply executed. In all cases, the code that's executed is expected to be valid as file input (see the section file-input in the Reference Manual). Be aware that the `nonlocal`, `yield`, and `return` statements may not be used outside of function definitions even within the context of code passed to the `exec()` function. The return value is `None`.

In all cases, if the optional parts are omitted, the code is executed in the current scope. If only *globals* is provided, it must be a dictionary (and not a subclass of dictionary), which will be used for both the global and the local variables. If *globals* and *locals* are given, they are used for the global and local variables, respectively.

1. Notez que l'analyseur n'accepte que des fin de lignes de style Unix. Si vous lisez le code depuis un fichier, assurez-vous d'utiliser la conversion de retours à la ligne pour convertir les fin de lignes Windows et Mac.

If provided, *locals* can be any mapping object. Remember that at the module level, globals and locals are the same dictionary.

> **Note**
>
> When `exec` gets two separate objects as *globals* and *locals*, the code will be executed as if it were embedded in a class definition. This means functions and classes defined in the executed code will not be able to access variables assigned at the top level (as the "top level" variables are treated as class variables in a class definition).

Si le dictionnaire *globals* ne contient pas de valeur pour la clé `__builtins__`, une référence au dictionnaire du module `builtins` y est inséré. Cela vous permet de contrôler quelles fonctions natives sont exposées au code exécuté en insérant votre propre dictionnaire `__builtins__` dans *globals* avant de le donner à `exec()`.

The *closure* argument specifies a closure--a tuple of cellvars. It's only valid when the *object* is a code object containing *free (closure) variables*. The length of the tuple must exactly match the length of the code object's `co_freevars` attribute.

Lève un *évènement d'audit* `exec` avec l'objet de code comme argument. Les événements de compilation de code peuvent également être levés.

> **Note**
>
> The built-in functions `globals()` and `locals()` return the current global and local namespace, respectively, which may be useful to pass around for use as the second and third argument to `exec()`.

> **Note**
>
> The default *locals* act as described for function `locals()` below. Pass an explicit *locals* dictionary if you need to see effects of the code on *locals* after function `exec()` returns.

Modifié dans la version 3.11 : ajout du paramètre *closure*.

Modifié dans la version 3.13 : The *globals* and *locals* arguments can now be passed as keywords.

Modifié dans la version 3.13 : The semantics of the default *locals* namespace have been adjusted as described for the `locals()` builtin.

filter(*function*, *iterable*)

Construit un itérateur depuis les éléments d'*iterable* pour lesquels *function* renvoie `True`. *iterable* peut aussi bien être une séquence, un conteneur qui prend en charge l'itération, ou un itérateur. Si *function* est `None`, la fonction identité est prise, c'est-à-dire que tous les éléments faux d'*iterable* sont supprimés.

Notez que `filter(function, iterable)` est l'équivalent du générateur `(item for item in iterable if function(item))` si *function* n'est pas `None`, et de `(item for item in iterable if item)` si *function* est `None`.

Voir `itertools.filterfalse()` pour la fonction complémentaire qui donne les éléments d'*iterable* pour lesquels *function* renvoie `False`.

class float(*number=0.0*, /)

class float(*string*, /)

Return a floating-point number constructed from a number or a string.

Exemples :

```
>>> float('+1.23')
1.23
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> float('   -12345\n')
-12345.0
>>> float('1e-003')
0.001
>>> float('+1E6')
1000000.0
>>> float('-Infinity')
-inf
```

If the argument is a string, it should contain a decimal number, optionally preceded by a sign, and optionally embedded in whitespace. The optional sign may be `'+'` or `'-'`; a `'+'` sign has no effect on the value produced. The argument may also be a string representing a NaN (not-a-number), or positive or negative infinity. More precisely, the input must conform to the *floatvalue* production rule in the following grammar, after leading and trailing whitespace characters are removed :

```
sign:           "+" | "-"
infinity:       "Infinity" | "inf"
nan:            "nan"
digit:          <a Unicode decimal digit, i.e. characters in Unicode general␣
↪category Nd>
digitpart:      digit (["_"] digit)*
number:         [digitpart] "." digitpart | digitpart ["."]
exponent:       ("e" | "E") [sign] digitpart
floatnumber:    number [exponent]
absfloatvalue:  floatnumber | infinity | nan
floatvalue:     [sign] absfloatvalue
```

Case is not significant, so, for example, "inf", "Inf", "INFINITY", and "iNfINity" are all acceptable spellings for positive infinity.

Otherwise, if the argument is an integer or a floating-point number, a floating-point number with the same value (within Python's floating-point precision) is returned. If the argument is outside the range of a Python float, an `OverflowError` will be raised.

Pour un objet Python général `x`, `float(x)` est délégué à `x.__float__()`. Si `__float__()` n'est pas défini alors il est délégué à `__index__()`.

See also `float.from_number()` which only accepts a numeric argument.

Sans argument, `0.0` est renvoyé.

Le type *float* est décrit dans *Types numériques — int, float, complex*.

Modifié dans la version 3.6 : les chiffres peuvent être groupés avec des tirets bas comme dans les expressions littérales.

Modifié dans la version 3.7 : Le paramètre est désormais un argument exclusivement positionnel.

Modifié dans la version 3.8 : délègue à `__index__()` si `__float__()` n'est pas définie.

format(*value*, *format_spec*="")

Convertit une valeur en sa représentation « formatée », contrôlée par *format_spec*. L'interprétation de *format_spec* dépend du type de la valeur. Cependant, il existe une syntaxe standard utilisée par la plupart des types natifs : *Mini-langage de spécification de format*.

Par défaut, *format_spec* est une chaîne vide. Dans ce cas, appeler cette fonction a généralement le même effet qu'appeler `str(value)`.

A call to `format(value, format_spec)` is translated to `type(value).__format__(value, format_spec)` which bypasses the instance dictionary when searching for the value's `__format__()` method. A `TypeError` exception is raised if the method search reaches `object` and the *format_spec* is non-empty, or if either the *format_spec* or the return value are not strings.

Modifié dans la version 3.4 : `object().__format__(format_spec)` lève `TypeError` si *format_spec* n'est pas une chaîne vide.

class frozenset(*iterable=set()*)

Renvoie un nouveau `frozenset`, dont les objets sont éventuellement tirés d'*iterable*. `frozenset` est une classe native. Voir `frozenset` et *Types d'ensembles — set, frozenset* pour la documentation sur cette classe.

Pour d'autres conteneurs, voyez les classes natives `set`, `list`, `tuple`, et `dict`, ainsi que le module `collections`.

getattr(*object*, *name*)

getattr(*object*, *name*, *default*)

Renvoie la valeur de l'attribut nommé *name* de l'objet *object*. *name* doit être une chaîne. Si la chaîne est le nom d'un des attributs de l'objet, le résultat est la valeur de cet attribut. Par exemple, `getattr(x, 'foobar')` est équivalent à `x.foobar`. Si l'attribut n'existe pas, mais que *default* est fourni, celui-ci est renvoyé. Sinon l'exception `AttributeError` est levée. *name* n'a pas besoin d'être un identifiant Python (voir `setattr()`).

> **Note**
>
> étant donné que la transformation des noms privés se produit au moment de la compilation, il faut modifier manuellement le nom d'un attribut privé (attributs avec deux traits de soulignement en tête) afin de le récupérer avec `getattr()`.

globals()

Renvoie le dictionnaire implémentant l'espace de nommage du module actuel. Pour le code dans les fonctions, il est défini lorsque la fonction est définie et reste le même quel que soit le moment où la fonction est appelée.

hasattr(*object*, *name*)

Les arguments sont : un objet et une chaîne de caractères. Le résultat est `True` si la chaîne est le nom d'un des attributs de l'objet, sinon `False` (l'implémentation appelle `getattr(object, name)` et regarde si une exception `AttributeError` a été levée).

hash(*object*)

Renvoie la valeur de hachage d'un objet (s'il en a une). Les valeurs de hachage sont des entiers. Elles sont utilisées pour comparer rapidement des clés de dictionnaire lors de leur recherche. Les valeurs numériques égales ont la même valeur de hachage (même si leurs types sont différents, comme pour `1` et `1.0`).

> **Note**
>
> For objects with custom `__hash__()` methods, note that `hash()` truncates the return value based on the bit width of the host machine.

help()

help(*request*)

Invoque le système d'aide natif (cette fonction est destinée à l'usage en mode interactif). Si aucun argument n'est fourni, le système d'aide démarre dans l'interpréteur. Si l'argument est une chaîne, celle-ci est recherchée comme le nom d'un module, d'une fonction, d'une classe, d'une méthode, d'un mot clé, ou d'un sujet de documentation, et une page d'aide est affichée sur la console. Si l'argument est d'un autre type, une page d'aide sur cet objet est générée.

Notez que si une barre oblique (/) apparaît dans la liste des paramètres d'une fonction lorsque vous appelez `help()`, cela signifie que les paramètres placés avant la barre oblique sont strictement positionnels. Pour plus d'informations, voir La FAQ sur les arguments positionnels.

Cette fonction est ajoutée à l'espace de nommage natif par le module `site`.

Modifié dans la version 3.4 : les changements aux modules `pydoc` et `inspect` rendent les signatures des appelables plus compréhensibles et cohérentes.

hex(*x*)

Convertit un entier en chaîne hexadécimale préfixée de "0x". Si *x* n'est pas un `int`, il doit définir une méthode `__index__()` qui renvoie un entier. Quelques exemples :

```
>>> hex(255)
'0xff'
>>> hex(-42)
'-0x2a'
```

Si vous voulez convertir un nombre entier en chaîne hexadécimale, en majuscule ou non, préfixée ou non, vous pouvez utiliser l'une des méthodes suivantes :

```
>>> '%#x' % 255, '%x' % 255, '%X' % 255
('0xff', 'ff', 'FF')
>>> format(255, '#x'), format(255, 'x'), format(255, 'X')
('0xff', 'ff', 'FF')
>>> f'{255:#x}', f'{255:x}', f'{255:X}'
('0xff', 'ff', 'FF')
```

Voir aussi `format()` pour plus d'informations.

Voir aussi `int()` pour convertir une chaîne hexadécimale en un entier (en affectant 16 à l'argument *base*).

> **Note**
>
> pour obtenir une représentation hexadécimale sous forme de chaîne d'un nombre à virgule flottante, utilisez la méthode `float.hex()`.

id(*object*)

Renvoie l'« identité » d'un objet. C'est un nombre entier garanti unique et constant pour cet objet durant sa durée de vie. Deux objets dont les durées de vie ne se chevauchent pas peuvent partager le même `id()`.

c'est l'adresse de l'objet en mémoire.

Lève un *évènement d'audit* `builtins.id` avec l'argument `id`.

input()

input(*prompt*)

Si l'argument *prompt* est donné, il est écrit sur la sortie standard sans le retour à la ligne final. La fonction lit ensuite une ligne sur l'entrée standard et la convertit en chaîne (supprimant le retour à la ligne final) qu'elle renvoie. Lorsque EOF est lu, `EOFError` est levée. Exemple :

```
>>> s = input('--> ')
--> Monty Python's Flying Circus
>>> s
"Monty Python's Flying Circus"
```

Si le module `readline` est chargé, `input()` l'utilisera pour fournir des fonctionnalités d'édition et d'historique élaborées.

Lève un *évènement d'audit* `builtins.input` avec l'argument `prompt` avant de lire l'entrée.

Lève un *événement d'audit* `builtins.input/result` avec le résultat après avoir lu avec succès l'entrée.

class int(*number=0, /*)

class int(*string, /, base=10*)

Return an integer object constructed from a number or a string, or return `0` if no arguments are given.

Exemples :

```
>>> int(123.45)
123
>>> int('123')
123
>>> int('   -12_345\n')
-12345
>>> int('FACE', 16)
64206
>>> int('0xface', 0)
64206
>>> int('01110011', base=2)
115
```

If the argument defines `__int__()`, `int(x)` returns `x.__int__()`. If the argument defines `__index__()`, it returns `x.__index__()`. For floating-point numbers, this truncates towards zero.

If the argument is not a number or if *base* is given, then it must be a string, `bytes`, or `bytearray` instance representing an integer in radix *base*. Optionally, the string can be preceded by `+` or `-` (with no space in between), have leading zeros, be surrounded by whitespace, and have single underscores interspersed between digits.

Une chaine représentant un entier en base *n* contient des chiffres, chacun représentant une valeur de 0 à n-1. Les valeurs 0 à 9 peuvent être représentées par n'importe lequel des chiffres décimaux Unicode. Les valeurs de 10 à 35 peuvent être représentées par `a` jusqu'à `z` (ou `A` à `Z`). La *base* par défaut est 10. Les valeurs autorisées pour *base* sont 0 et 2 à 36. Les littéraux en base 2, 8, et 16 peuvent être préfixés avec `0b`/`0B`, `0o`/`0O`, ou `0x`/`0X` tout comme les littéraux dans le code. Fournir 0 comme *base* demande d'interpréter exactement comme un entier littéral dans du code Python, donc la base sera 2, 8, 10, ou 16 en fonction du préfixe. Indiquer 0 comme base interdit les zéros en tête, ainsi `int('010', 0)` n'est pas légal, alors que `int('010')` l'est tout comme `int('010', 8)`.

Le type des entiers est décrit dans *Types numériques — int, float, complex*.

Modifié dans la version 3.4 : si *base* n'est pas une instance d'`int` et que *base* a une méthode `base.__index__`, cette méthode est appelée pour obtenir un entier pour cette base. Les versions précédentes utilisaient `base.__int__` au lieu de `base.__index__`.

Modifié dans la version 3.6 : les chiffres peuvent être groupés avec des tirets bas comme dans les expressions littérales.

Modifié dans la version 3.7 : The first parameter is now positional-only.

Modifié dans la version 3.8 : Revient à `__index__()` si `__int__()` n'est pas définie.

Modifié dans la version 3.11 : `int` string inputs and string representations can be limited to help avoid denial of service attacks. A `ValueError` is raised when the limit is exceeded while converting a string to an `int` or when converting an `int` into a string would exceed the limit. See the *integer string conversion length limitation* documentation.

Modifié dans la version 3.14 : `int()` no longer delegates to the `__trunc__()` method.

isinstance(*object*, *classinfo*)

Renvoie `True` si *object* est une instance de *classinfo*, ou d'une de ses classes filles, directe, indirecte, ou *abstraite*. Si *object* n'est pas un objet du type donné, la fonction renvoie toujours `False`. Si *classinfo* est un *n*-uplet de types (ou récursivement, d'autres *n*-uplets) ou une *union de types* différents, renvoie `True` si *object* est une instance de n'importe quel de ces types. Si *classinfo* n'est ni un type ni un *n*-uplet de types (et récursivement), une exception `TypeError` est levée. `TypeError` ne peut pas être levée pour un type invalide si une des vérifications précédentes réussit.

Modifié dans la version 3.10 : *classinfo* peut être une *union de types*.

issubclass(*class*, *classinfo*)

Renvoie `True` si *class* est une classe fille (directe, indirecte ou *abstraite*) de *classinfo*. Une classe est considérée sous-classe d'elle-même. *classinfo* peut être un *n*-uplet de classes (ou récursivement, d'autres *n*-uplets) ou une

union de types, dans ce cas elle renvoie `True` si *class* est une sous-classe d'une partie de *classinfo*. Dans tous les autres cas, une `TypeError` est levée.

Modifié dans la version 3.10 : *classinfo* peut être une *union de types*.

iter(*object*)

iter(*object*, *sentinel*)

Return an *iterator* object. The first argument is interpreted very differently depending on the presence of the second argument. Without a second argument, *object* must be a collection object which supports the *iterable* protocol (the `__iter__()` method), or it must support the sequence protocol (the `__getitem__()` method with integer arguments starting at `0`). If it does not support either of those protocols, `TypeError` is raised. If the second argument, *sentinel*, is given, then *object* must be a callable object. The iterator created in this case will call *object* with no arguments for each call to its `__next__()` method; if the value returned is equal to *sentinel*, `StopIteration` will be raised, otherwise the value will be returned.

Voir aussi *Les types itérateurs*.

Une autre application utile de la deuxième forme de `iter()` est de construire un lecteur par blocs. Par exemple, lire des blocs de taille fixe d'une base de donnée binaire jusqu'à ce que la fin soit atteinte :

```
from functools import partial
with open('mydata.db', 'rb') as f:
    for block in iter(partial(f.read, 64), b''):
        process_block(block)
```

len(*s*)

Renvoie la longueur (nombre d'éléments) d'un objet. L'argument peut être une séquence (telle qu'une chaîne de caractères ou d'octets, un *n*-uplet, une liste ou un intervalle) ou une collection (telle qu'un dictionnaire, un ensemble ou un ensemble figé).

`len` lève une `OverflowError` sur des longueurs supérieures à `sys.maxsize`, par exemple pour `range(2 ** 100)`.

class list

class list(*iterable*)

Contrairement aux apparences, `list` n'est pas une fonction mais un type séquentiel mutable, comme décrit dans *Listes* et *Types séquentiels — list, tuple, range*.

locals()

Return a mapping object representing the current local symbol table, with variable names as the keys, and their currently bound references as the values.

At module scope, as well as when using `exec()` or `eval()` with a single namespace, this function returns the same namespace as `globals()`.

At class scope, it returns the namespace that will be passed to the metaclass constructor.

When using `exec()` or `eval()` with separate local and global arguments, it returns the local namespace passed in to the function call.

In all of the above cases, each call to `locals()` in a given frame of execution will return the *same* mapping object. Changes made through the mapping object returned from `locals()` will be visible as assigned, reassigned, or deleted local variables, and assigning, reassigning, or deleting local variables will immediately affect the contents of the returned mapping object.

In an *optimized scope* (including functions, generators, and coroutines), each call to `locals()` instead returns a fresh dictionary containing the current bindings of the function's local variables and any nonlocal cell references. In this case, name binding changes made via the returned dict are *not* written back to the corresponding local variables or nonlocal cell references, and assigning, reassigning, or deleting local variables and nonlocal cell references does *not* affect the contents of previously returned dictionaries.

Calling `locals()` as part of a comprehension in a function, generator, or coroutine is equivalent to

calling it in the containing scope, except that the comprehension's initialised iteration variables will be included. In other scopes, it behaves as if the comprehension were running as a nested function.

Calling `locals()` as part of a generator expression is equivalent to calling it in a nested generator function.

Modifié dans la version 3.12 : The behaviour of `locals()` in a comprehension has been updated as described in **PEP 709**.

Modifié dans la version 3.13 : As part of **PEP 667**, the semantics of mutating the mapping objects returned from this function are now defined. The behavior in *optimized scopes* is now as described above. Aside from being defined, the behaviour in other scopes remains unchanged from previous versions.

map(*function*, *iterable*, */*, **iterables*, *strict=False*)

Return an iterator that applies *function* to every item of *iterable*, yielding the results. If additional *iterables* arguments are passed, *function* must take that many arguments and is applied to the items from all iterables in parallel. With multiple iterables, the iterator stops when the shortest iterable is exhausted. If *strict* is `True` and one of the iterables is exhausted before the others, a `ValueError` is raised. For cases where the function inputs are already arranged into argument tuples, see `itertools.starmap()`.

Modifié dans la version 3.14 : Added the *strict* parameter.

max(*iterable*, ***, *key=None*)

max(*iterable*, ***, *default*, *key=None*)

max(*arg1*, *arg2*, **args*, *key=None*)

Renvoie le plus grand élément d'un itérable, ou l'argument le plus grand parmi au moins deux arguments.

Si un seul argument positionnel est fourni, il doit être *itérable*. Le plus grand élément de l'itérable est renvoyé. Si au moins deux arguments positionnels sont fournis, l'argument le plus grand sera renvoyé.

Elle accepte deux arguments nommés optionnels. L'argument *key* spécifie une fonction à un argument permettant de trier comme pour `list.sort()`. L'argument *default* fournit quant à lui un objet à donner si l'itérable fourni est vide. Si l'itérable est vide et que *default* n'est pas fourni, `ValueError` est levée.

Si plusieurs éléments représentent la plus grande valeur, le premier rencontré est renvoyé. C'est cohérent avec d'autres outils préservant une stabilité lors du tri, tels que `sorted(iterable, key=keyfunc, reverse=True)[0]` et `heapq.nlargest(1, iterable, key=keyfunc)`.

Modifié dans la version 3.4 : Added the *default* keyword-only parameter.

Modifié dans la version 3.8 : l'argument *key* peut être `None`.

class memoryview(*object*)

Renvoie une « vue mémoire » (*memory view*) créée depuis l'argument. Voir *Vues mémoire* pour plus d'informations.

min(*iterable*, ***, *key=None*)

min(*iterable*, ***, *default*, *key=None*)

min(*arg1*, *arg2*, **args*, *key=None*)

Renvoie le plus petit élément d'un itérable ou le plus petit d'au moins deux arguments.

Si un seul argument est fourni, il doit être *itérable*. Le plus petit élément de l'itérable est renvoyé. Si au moins deux arguments positionnels sont fournis, le plus petit argument positionnel est renvoyé.

Elle accepte deux arguments nommés optionnels. L'argument *key* spécifie une fonction à un argument permettant de trier comme pour `list.sort()`. L'argument *default* fournit quant à lui un objet à donner si l'itérable fourni est vide. Si l'itérable est vide et que *default* n'est pas fourni, `ValueError` est levée.

Si plusieurs éléments sont minimaux, la fonction renvoie le premier rencontré. C'est cohérent avec d'autres outils préservant une stabilité lors du tri, tels que `sorted(iterable, key=keyfunc)[0]` et `heapq.nsmallest(1, iterable, key=keyfunc)`.

Modifié dans la version 3.4 : Added the *default* keyword-only parameter.

Modifié dans la version 3.8 : l'argument *key* peut être `None`.

next (*iterator*)

next (*iterator*, *default*)

Donne l'élément suivant de l'*itérateur* en appelant sa méthode `__next__()`. Si *default* est fourni, il est renvoyé quand l'itérateur est épuisé, sinon une `StopIteration` est levée.

class object

This is the ultimate base class of all other classes. It has methods that are common to all instances of Python classes. When the constructor is called, it returns a new featureless object. The constructor does not accept any arguments.

> **Note**
>
> `object` instances do *not* have `__dict__` attributes, so you can't assign arbitrary attributes to an instance of `object`.

oct (*x*)

Convertit un entier en sa représentation octale dans une chaîne préfixée de "0o". Le résultat est une expression Python valide. Si *x* n'est pas un objet `int`, il doit définir une méthode `__index__()` qui donne un entier, par exemple :

```
>>> oct(8)
'0o10'
>>> oct(-56)
'-0o70'
```

Si vous voulez convertir un nombre entier en une chaîne octale, avec ou sans le préfixe `0o`, vous pouvez utiliser l'une des méthodes suivantes.

```
>>> '%#o' % 10, '%o' % 10
('0o12', '12')
>>> format(10, '#o'), format(10, 'o')
('0o12', '12')
>>> f'{10:#o}', f'{10:o}'
('0o12', '12')
```

Voir aussi `format()` pour plus d'informations.

open (*file*, *mode='r'*, *buffering=-1*, *encoding=None*, *errors=None*, *newline=None*, *closefd=True*, *opener=None*)

Ouvre *file* et donne un *objet fichier* correspondant. Si le fichier ne peut pas être ouvert, une `OSError` est levée. Voir tut-files pour plus d'exemples d'utilisation de cette fonction.

file est un *objet simili-chemin* donnant le chemin (absolu ou relatif au répertoire courant) du fichier à ouvrir ou un nombre entier représentant le descripteur de fichier à envelopper (si un descripteur de fichier est donné, il sera fermé en même temps que l'objet d'entrée-sortie renvoyé, sauf si *closefd* est mis à `False`).

mode is an optional string that specifies the mode in which the file is opened. It defaults to `'r'` which means open for reading in text mode. Other common values are `'w'` for writing (truncating the file if it already exists), `'x'` for exclusive creation, and `'a'` for appending (which on *some* Unix systems, means that *all* writes append to the end of the file regardless of the current seek position). In text mode, if *encoding* is not specified the encoding used is platform-dependent : `locale.getencoding()` is called to get the current locale encoding. (For reading and writing raw bytes use binary mode and leave *encoding* unspecified.) The available modes are :

Caractère	Signification
`'r'`	ouvre en lecture (par défaut)
`'w'`	ouvre en écriture, en effaçant le contenu du fichier
`'x'`	ouvre pour une création exclusive, échouant si le fichier existe déjà
`'a'`	ouvre en écriture, ajoutant à la fin du fichier s'il existe
`'b'`	mode binaire
`'t'`	mode texte (par défaut)
`'+'`	ouvre en modification (lecture et écriture)

Le mode par défaut est `'r'` (ouverture pour lire du texte, un synonyme pour `'rt'`). Les modes `'w+'` et `'w+b'` ouvrent et vident le fichier. Les modes `'r+'` et `'r+b'` ouvrent le fichier sans le vider.

Comme mentionné dans *Aperçu*, Python fait la différence entre les entrées-sorties binaires et textes. Les fichiers ouverts en mode binaire (avec `'b'` dans *mode*) donnent leur contenu sous forme de `bytes` sans décodage. En mode texte (par défaut, ou lorsque `'t'` est dans le *mode*), le contenu du fichier est donné sous forme de `str`, les octets ayant été décodés au préalable en utilisant un encodage déduit de l'environnement ou *encoding* s'il est donné.

> **Note**
>
> Python ne dépend pas de la représentation du fichier texte du système sous-jacent. Tout le traitement est effectué par Python lui-même, et est ainsi indépendant de la plate-forme.

buffering is an optional integer used to set the buffering policy. Pass 0 to switch buffering off (only allowed in binary mode), 1 to select line buffering (only usable when writing in text mode), and an integer > 1 to indicate the size in bytes of a fixed-size chunk buffer. Note that specifying a buffer size this way applies for binary buffered I/O, but `TextIOWrapper` (i.e., files opened with `mode='r+'`) would have another buffering. To disable buffering in `TextIOWrapper`, consider using the `write_through` flag for `io.TextIOWrapper.reconfigure()`. When no *buffering* argument is given, the default buffering policy works as follows :

— les fichiers binaires sont mis dans un tampon de taille fixe, dont la taille est choisie par une heuristique essayant de déterminer la taille des blocs du système sous-jacent, ou en utilisant par défaut `io.DEFAULT_BUFFER_SIZE`. Sur de nombreux systèmes, le tampon est de 4096 ou 8192 octets ;
— les fichiers texte « interactifs » (fichiers pour lesquels `io.IOBase.isatty()` renvoie `True`) utilisent un tampon ligne par ligne. Les autres fichiers textes sont traités comme les fichiers binaires.

encoding est le nom de l'encodage utilisé pour encoder ou décoder le fichier. Il doit seulement être utilisé en mode texte. L'encodage par défaut dépend de la plateforme (ce que renvoie `locale.getencoding()`), mais n'importe quel *encodage de texte* pris en charge par Python peut être utilisé. Voir `codecs` pour une liste des encodages pris en charge.

errors est une chaîne facultative spécifiant comment les erreurs d'encodage et de décodage sont gérées, ce n'est pas utilisable en mode binaire. De nombreux gestionnaires d'erreurs standards sont disponibles (listés sous *Gestionnaires d'erreurs*), aussi, tout nom de gestionnaire d'erreur enregistré avec `codecs.register_error()` est aussi un argument valide. Les noms standards sont :

— `'strict'` pour lever une `ValueError` si une erreur d'encodage est rencontrée. La valeur par défaut, `None`, a le même effet.
— `'ignore'` ignore les erreurs. Notez qu'ignorer les erreurs d'encodage peut mener à des pertes de données.
— `'replace'` insère un marqueur de substitution (tel que `'?'`) en place des données mal formées.
— `'surrogateescape'` représentera tous les octets incorrects comme des points de code de substitution de l'intervalle bas U+DC80 à U+DCFF. Ces points de code de substitution sont ensuite retransformés dans les mêmes octets lorsque le gestionnaire d'erreurs `surrogateescape` est utilisé pour l'écriture des données. C'est utile pour traiter des fichiers d'un encodage inconnu.
— `'xmlcharrefreplace'` is only supported when writing to a file. Characters not supported by the encoding are replaced with the appropriate XML character reference `&#nnn;`.
— `'backslashreplace'` remplace les données mal formées par des séquences d'échappement Python (utilisant des barres obliques inverses).
— `'namereplace'` (aussi supporté lors de l'écriture) remplace les caractères non gérés par des séquences

d'échappement `\N{...}`.

newline contrôle comment interpréter les retours à la ligne. Il peut être `None`, `''`, `'\n'`, `'\r'`, et `'\r\n'`. Il fonctionne comme suit :

— Lors de la lecture, si *newline* est `None`, le mode *universal newlines* est activé. Les lignes lues peuvent se terminer par `'\n'`, `'\r'`, ou `'\r\n'`, et sont remplacées par `'\n'`, avant d'être renvoyées à l'appelant. S'il vaut `''`, le mode *universal newline* est activé mais les fins de ligne ne sont pas remplacées. S'il a n'importe quelle autre valeur autorisée, les lignes sont seulement terminées par la chaîne donnée, qui est rendue telle quelle.

— Lors de l'écriture, si *newline* est `None`, chaque `'\n'` est remplacé par le séparateur de lignes par défaut du système `os.linesep`. Si *newline* est `''` ou `'\n'` aucun remplacement n'est effectué. Si *newline* est un autre caractère valide, chaque `'\n'` sera remplacé par la chaîne donnée.

Si *closefd* est `False` et qu'un descripteur de fichier est fourni plutôt qu'un nom de fichier, le descripteur de fichier sera laissé ouvert lorsque le fichier sera fermé. Si un nom de fichier est donné, *closefd* doit rester `True` (la valeur par défaut) ; sinon, une erreur est levée.

Un *opener* personnalisé peut être utilisé en fournissant un appelable comme *opener*. Le descripteur de fichier de cet objet fichier sera alors obtenu en appelant *opener* avec (*file*, *flags*). *opener* doit renvoyer un descripteur de fichier ouvert (fournir `os.open` en tant qu'*opener* aura le même effet que donner `None`).

Il n'est *pas possible d'hériter du descripteur de fichier* nouvellement créé.

L'exemple suivant utilise le paramètre *dir_fd* de la fonction `os.open()` pour ouvrir un fichier relatif au dossier courant :

```
>>> import os
>>> dir_fd = os.open('somedir', os.O_RDONLY)
>>> def opener(path, flags):
...     return os.open(path, flags, dir_fd=dir_fd)
...
>>> with open('spamspam.txt', 'w', opener=opener) as f:
...     print('This will be written to somedir/spamspam.txt', file=f)
...
>>> os.close(dir_fd)  # don't leak a file descriptor
```

Le type d'*objet fichier* renvoyé par la fonction `open()` dépend du mode. Lorsque `open()` est utilisé pour ouvrir un fichier en mode texte (`w`, `r`, `wt`, `rt`, etc.), il renvoie une classe fille de `io.TextIOBase` (spécifiquement : `io.TextIOWrapper`). Lors de l'ouverture d'un fichier en mode binaire avec tampon, la classe renvoyée sera une fille de `io.BufferedIOBase`. La classe exacte varie : en lecture en mode binaire elle renvoie une `io.BufferedReader`, en écriture et ajout en mode binaire c'est une `io.BufferedWriter`, et en lecture-écriture, c'est une `io.BufferedRandom`. Lorsque le tampon est désactivé, le flux brut, une classe fille de `io.RawIOBase`, `io.FileIO` est renvoyée.

Consultez aussi les modules de gestion de fichiers tels que `fileinput`, `io` (où `open()` est déclarée), `os`, `os.path`, `tempfile`, et `shutil`.

Raises an *auditing event* `open` with arguments `path`, `mode`, `flags`.

Les arguments `mode` et `flags` peuvent avoir été modifiés ou déduits de l'appel original.

Modifié dans la version 3.3 :

— ajout du paramètre *opener*.

— ajout du mode `'x'`.

— `IOError` était normalement levée, elle est maintenant un alias de `OSError`.

— `FileExistsError` est maintenant levée si le fichier ouvert en mode création exclusive (`'x'`) existe déjà.

Modifié dans la version 3.4 :

— Il n'est plus possible d'hériter de *file*.

Modifié dans la version 3.5 :

— Si l'appel système est interrompu et que le gestionnaire de signal ne lève aucune exception, la fonction réessaye l'appel système au lieu de lever une `InterruptedError` (voir la **PEP 475** pour la justification).

— ajout du gestionnaire d'erreurs `'namereplace'`.

Modifié dans la version 3.6 :

— prise en charge des objets implémentant `os.PathLike`.

— Sous Windows, ouvrir un *buffer* du terminal peut renvoyer une sous-classe de `io.RawIOBase` autre que `io.FileIO`.

Modifié dans la version 3.11 : le mode `'U'` a été enlevé.

ord(*c*)

Renvoie le nombre entier représentant le code Unicode du caractère représenté par la chaîne donnée. Par exemple, `ord('a')` renvoie le nombre entier `97` et `ord('€')` (symbole euro) renvoie `8364`. Il s'agit de la réciproque de `chr()`.

pow(*base*, *exp*, *mod=None*)

Renvoie *base* puissance *exp* et, si *mod* est présent, donne *base* puissance *exp* modulo *mod* (calculé de manière plus efficiente que `pow(base, exp) % mod`). La forme à deux arguments `pow(base, exp)` est équivalente à l'opérateur puissance : `base**exp`.

The arguments must have numeric types. With mixed operand types, the coercion rules for binary arithmetic operators apply. For `int` operands, the result has the same type as the operands (after coercion) unless the second argument is negative ; in that case, all arguments are converted to float and a float result is delivered. For example, `pow(10, 2)` returns `100`, but `pow(10, -2)` returns `0.01`. For a negative base of type `int` or `float` and a non-integral exponent, a complex result is delivered. For example, `pow(-9, 0.5)` returns a value close to `3j`. Whereas, for a negative base of type `int` or `float` with an integral exponent, a float result is delivered. For example, `pow(-9, 2.0)` returns `81.0`.

Pour des opérandes *base* et *exp* de type `int`, si *mod* est présent, *mod* doit également être de type entier et *mod* doit être non nul. Si *mod* est présent et que *exp* est négatif, *base* et *mod* doivent être premiers entre eux. Dans ce cas, `pow(inv_base, -exp, mod)` est renvoyé, où *inv_base* est un inverse de *base* modulo *mod*.

Voici un exemple de calcul d'un inverse de `38` modulo `97` :

```
>>> pow(38, -1, mod=97)
23
>>> 23 * 38 % 97 == 1
True
```

Modifié dans la version 3.8 : pour les opérandes `int`, la forme à trois arguments de `pow` permet maintenant au deuxième argument d'être négatif, permettant le calcul des inverses modulaires.

Modifié dans la version 3.8 : autorise les arguments nommés. Auparavant, seuls les arguments positionnels étaient autorisés.

print(**objects*, *sep=' '*, *end='\n'*, *file=None*, *flush=False*)

Écrit *objects* dans le flux texte *file*, séparés par *sep* et suivis de *end*. Les arguments *sep*, *end*, *file*, et *flush*, s'ils sont présents, doivent être passés en arguments nommés.

Tous les arguments positionnels sont convertis en chaîne comme le fait `str()`, puis écrits sur le flux, séparés par *sep* et terminés par *end*. *sep* et *end* doivent être des chaînes, ou `None`, indiquant de prendre les valeurs par défaut. Si aucun *objects* n'est donné `print()` écrit seulement *end*.

L'argument *file* doit être un objet avec une méthode `write(string)` ; s'il n'est pas fourni, ou vaut `None`, `sys.stdout` est utilisé. Puisque les arguments affichés sont convertis en chaîne, `print()` ne peut pas être utilisée avec des fichiers ouverts en mode binaire. Pour ceux-ci utilisez plutôt `file.write(...)`.

Que la sortie utilise un tampon ou non est souvent décidé par *file*. Cependant, si l'argument *flush* est vrai, le tampon du flux est vidé explicitement.

Modifié dans la version 3.3 : ajout de l'argument nommé *flush*.

class property(*fget=None*, *fset=None*, *fdel=None*, *doc=None*)

Renvoie un attribut propriété.

fget est une fonction permettant d'obtenir la valeur d'un attribut. *fset* est une fonction pour en définir la valeur. *fdel* quant à elle permet de supprimer la valeur d'un attribut, et *doc* crée une chaîne de documentation (*docstring*) pour l'attribut.

Une utilisation courante est de définir un attribut managé `x` :

```
class C:
    def __init__(self):
        self._x = None

    def getx(self):
        return self._x

    def setx(self, value):
        self._x = value

    def delx(self):
        del self._x

    x = property(getx, setx, delx, "I'm the 'x' property.")
```

Si *c* est une instance de *C*, `c.x` appelle l'accesseur (*getter* en anglais), `c.x = value` invoque le mutateur (*setter*), et `del x` le destructeur (*deleter*).

Si elle est fournie, *doc* est la chaîne de documentation de l'attribut. Autrement la propriété copie celle de *fget* (si elle existe). Cela rend possible la création de propriétés en lecture seule en utilisant simplement `property()` comme *décorateur* :

```
class Parrot:
    def __init__(self):
        self._voltage = 100000

    @property
    def voltage(self):
        """Get the current voltage."""
        return self._voltage
```

The `@property` decorator turns the `voltage()` method into a "getter" for a read-only attribute with the same name, and it sets the docstring for *voltage* to "Get the current voltage."

@getter

@setter

@deleter

A property object has `getter`, `setter`, and `deleter` methods usable as decorators that create a copy of the property with the corresponding accessor function set to the decorated function. This is best explained with an example :

```
class C:
    def __init__(self):
        self._x = None

    @property
    def x(self):
        """I'm the 'x' property."""
        return self._x

    @x.setter
    def x(self, value):
        self._x = value

    @x.deleter
    def x(self):
        del self._x
```

Ce code est l'exact équivalent du premier exemple. Soyez attentifs à bien donner aux fonctions additionnelles le même nom que la propriété (`x` dans ce cas).

L'objet propriété renvoyé possède aussi les attributs `fget`, `fset` et `fdel` correspondants aux arguments du constructeur.

Modifié dans la version 3.5 : les chaînes de documentation des objets *property* sont maintenant en lecture-écriture.

__name__

Attribute holding the name of the property. The name of the property can be changed at runtime.

Ajouté dans la version 3.13.

class range(*stop*)

class range(*start*, *stop*, *step=1*)

Contrairement aux apparences, `range` n'est pas une fonction mais un type de séquence immuable, comme décrit dans *Ranges* et *Types séquentiels — list, tuple, range*.

repr(*object*)

Return a string containing a printable representation of an object. For many types, this function makes an attempt to return a string that would yield an object with the same value when passed to `eval()` ; otherwise, the representation is a string enclosed in angle brackets that contains the name of the type of the object together with additional information often including the name and address of the object. A class can control what this function returns for its instances by defining a `__repr__()` method. If `sys.displayhook()` is not accessible, this function will raise `RuntimeError`.

This class has a custom representation that can be evaluated :

```
class Person:
   def __init__(self, name, age):
      self.name = name
      self.age = age

   def __repr__(self):
      return f"Person('{self.name}', {self.age})"
```

reversed(*seq*)

Return a reverse *iterator*. *seq* must be an object which has a `__reversed__()` method or supports the sequence protocol (the `__len__()` method and the `__getitem__()` method with integer arguments starting at `0`).

round(*number*, *ndigits=None*)

Renvoie *number* arrondi avec une précision de *ndigits* chiffres après la virgule. Si *ndigits* est omis (ou est `None`), l'entier le plus proche est renvoyé.

Pour les types natifs prenant en charge `round()`, les valeurs sont arrondies au multiple de 10 puissance moins *ndigits*, si deux multiples sont équidistants, l'arrondi se fait vers la valeur paire (par exemple `round(0.5)` et `round(-0.5)` valent tous les deux `0`, et `round(1.5)` vaut `2`). *ndigits* accepte tout nombre entier (positif, zéro, ou négatif). La valeur renvoyée est un entier si *ndigits* n'est pas donné (ou est `None`). Sinon elle est du même type que *number*.

Pour tout autre objet Python `number`, `round` délègue à `number.__round__`.

> **Note**
>
> le comportement de `round()` avec les nombres à virgule flottante peut être surprenant. Par exemple `round(2.675, 2)` donne `2.67` au lieu de `2.68`. Ce n'est pas un bogue, mais dû au fait que la plupart des fractions de décimaux ne peuvent pas être représentés exactement en nombre a virgule flottante. Voir tut-fp-issues pour plus d'information.

class set

class set(*iterable*)

Renvoie un nouvel `ensemble`, dont les éléments sont extraits d'*iterable* s'il est fourni. `set` est une classe native. Voir `set` et *Types d'ensembles — set, frozenset* pour la documentation de cette classe.

D'autres conteneurs existent, comme : `frozenset`, `list`, `tuple`, et `dict`, ainsi que le module `collections`.

setattr(*object*, *name*, *value*)

C'est le complément de `getattr()`. Les arguments sont : un objet, une chaîne et une valeur de type arbitraire. La chaîne *name* peut désigner un attribut existant ou un nouvel attribut. La fonction assigne la valeur à l'attribut, si l'objet l'autorise. Par exemple, `setattr(x, 'foobar', 123)` équivaut à `x.foobar = 123`.

name n'a pas besoin d'être un identifiant Python tel que défini dans identifiers sauf si l'objet choisit de l'imposer, par exemple en personnalisant `__getattribute__()` ou *via* `__slots__`. Un attribut dont le nom n'est pas un identifiant ne sera pas accessible en utilisant la notation pointée, mais est accessible *via* `getattr()` etc.

> **Note**
>
> étant donné que la transformation des noms privés se produit au moment de la compilation, il faut modifier manuellement le nom d'un attribut privé (attributs avec deux traits de soulignement en tête) afin de le définir avec `setattr()`.

class slice(*stop*)

class slice(*start*, *stop*, *step=None*)

Return a *slice* object representing the set of indices specified by `range(start, stop, step)`. The *start* and *step* arguments default to `None`.

start

stop

step

Slice objects have read-only data attributes `start`, `stop`, and `step` which merely return the argument values (or their default). They have no other explicit functionality ; however, they are used by NumPy and other third-party packages.

Slice objects are also generated when extended indexing syntax is used. For example : `a[start:stop:step]` or `a[start:stop, i]`. See `itertools.islice()` for an alternate version that returns an *iterator*.

Modifié dans la version 3.12 : Slice objects are now *hashable* (provided `start`, `stop`, and `step` are hashable).

sorted(*iterable*, /, *, *key=None*, *reverse=False*)

Renvoie une nouvelle liste triée depuis les éléments d'*iterable*.

Possède deux arguments optionnels qui doivent être spécifiés par arguments nommés.

key spécifie une fonction à un argument utilisée pour extraire une clé de comparaison de chaque élément de l'itérable (par exemple, `key=str.lower`). La valeur par défaut est `None` (compare les éléments directement).

reverse est une valeur booléenne. Si elle est `True`, la liste d'éléments est triée comme si toutes les comparaisons étaient inversées.

Utilisez `functools.cmp_to_key()` pour convertir l'ancienne notation *cmp* en une fonction *key*.

La fonction native `sorted()` est garantie stable. Un tri est stable s'il garantit de ne pas changer l'ordre relatif des éléments égaux entre eux. C'est utile pour trier en plusieurs passes (par exemple par direction puis par salaire).

L'algorithme de tri utilise uniquement l'opérateur < pour comparer les éléments. Alors que la définition d'une méthode `__lt__()` est suffisante pour trier, la **PEP 8** recommande que les six comparaisons riches soient implémentées. Cela contribue à éviter les bogues lors de l'utilisation des mêmes données avec d'autres outils

de tri (tels que `max()`) qui reposent sur une méthode sous-jacente différente. L'implémentation des six comparaisons permet également d'éviter toute confusion lors de comparaisons de types mixtes qui peuvent appeler la méthode `__gt__()`.

Pour des exemples de tris et un bref tutoriel, consultez sortinghowto.

@staticmethod

Transforme une méthode en méthode statique.

Une méthode statique ne reçoit pas de premier argument implicitement. Voici comment déclarer une méthode statique :

```
class C:
    @staticmethod
    def f(arg1, arg2, argN): ...
```

La forme `@staticmethod` est un *décorateur* de fonction. Consultez function pour plus de détails.

A static method can be called either on the class (such as `C.f()`) or on an instance (such as `C().f()`). Moreover, the static method *descriptor* is also callable, so it can be used in the class definition (such as `f()`).

Les méthodes statiques en Python sont similaires à celles que l'on trouve en Java ou en C++. Consultez `classmethod()` pour une variante utile permettant de créer des constructeurs alternatifs.

Comme pour tous les décorateurs, il est possible d'appeler `staticmethod` comme une simple fonction, et faire quelque chose de son résultat. Ça peut être nécessaire dans le cas où vous souhaitez une référence à la fonction depuis le corps d'une classe, et voulez éviter sa transformation en méthode d'instance. Dans ce cas, faites comme suit :

```
def regular_function():
    ...

class C:
    method = staticmethod(regular_function)
```

Pour plus d'informations sur les méthodes statiques, consultez types.

Modifié dans la version 3.10 : Static methods now inherit the method attributes (`__module__`, `__name__`, `__qualname__`, `__doc__` and `__annotations__`), have a new `__wrapped__` attribute, and are now callable as regular functions.

class str(*object=''*)

class str(*object=b'', encoding='utf-8', errors='strict'*)

Renvoie une version d'*object* sous forme de `str`. Voir `str()` pour plus de détails.

`str` est la *classe* native des chaînes de caractères. Pour des informations générales à propos des chaînes, consultez *Type Séquence de Texte — str*.

sum(*iterable, /, start=0*)

Additionne *start* et les éléments d'un *iterable* de gauche à droite et en donne le total. Les éléments de l'*iterable* sont normalement des nombres, et la valeur de *start* ne peut pas être une chaîne de caractères.

For some use cases, there are good alternatives to `sum()`. The preferred, fast way to concatenate a sequence of strings is by calling `''.join(sequence)`. To add floating-point values with extended precision, see `math.fsum()`. To concatenate a series of iterables, consider using `itertools.chain()`.

Modifié dans la version 3.8 : le paramètre *start* peut être passé comme un argument nommé.

Modifié dans la version 3.12 : Summation of floats switched to an algorithm that gives higher accuracy and better commutativity on most builds.

Modifié dans la version 3.14 : Added specialization for summation of complexes, using same algorithm as for summation of floats.

class super

class super(*type*, *object_or_type=None*)

Renvoie un objet mandataire (*proxy object* en anglais) déléguant les appels de méthode à une classe parente ou sœur de *type*. C'est utile pour accéder aux méthodes héritées qui ont été remplacées dans une classe.

object_or_type détermine quel *ordre de résolution des méthodes* est utilisé pour la recherche. La recherche commence à partir de la classe qui suit immédiatement le *type*.

For example, if `__mro__` of *object_or_type* is `D -> B -> C -> A -> object` and the value of *type* is `B`, then `super()` searches `C -> A -> object`.

The `__mro__` attribute of the class corresponding to *object_or_type* lists the method resolution search order used by both `getattr()` and `super()`. The attribute is dynamic and can change whenever the inheritance hierarchy is updated.

Si le second argument est omis, l'objet *super* obtenu n'est pas lié. Si le second argument est un objet, `isinstance(obj, type)` doit être vrai. Si le second argument est un type, `issubclass(type2, type)` doit être vrai (c'est utile pour les méthodes de classe).

When called directly within an ordinary method of a class, both arguments may be omitted ("zero-argument `super()`"). In this case, *type* will be the enclosing class, and *obj* will be the first argument of the immediately enclosing function (typically `self`). (This means that zero-argument `super()` will not work as expected within nested functions, including generator expressions, which implicitly create nested functions.)

Il existe deux autres cas d'usage typiques pour *super*. Dans une hiérarchie de classes à héritage simple, *super* peut être utilisée pour obtenir la classe parente sans avoir à la nommer explicitement, rendant le code plus maintenable. Cet usage se rapproche de l'usage de *super* dans d'autres langages de programmation.

Le second est la gestion d'héritage multiple coopératif dans un environnement d'exécution dynamique. Cet usage est unique à Python, il ne se retrouve ni dans les langages compilés statiquement, ni dans les langages ne gérant que l'héritage simple. Cela rend possible d'implémenter un héritage en diamant dans lequel plusieurs classes parentes implémentent la même méthode. Une bonne conception implique que ces implémentations doivent avoir la même signature lors de leur appel dans tous les cas (parce que l'ordre des appels est déterminée à l'exécution, parce que l'ordre s'adapte aux changements dans la hiérarchie, et parce que l'ordre peut inclure des classes sœurs inconnues avant l'exécution).

Dans tous les cas, un appel typique à une classe parente ressemble à :

```
class C(B):
    def method(self, arg):
        super().method(arg)     # This does the same thing as:
                                # super(C, self).method(arg)
```

En plus de la recherche de méthodes, `super()` fonctionne également pour la recherche d'attributs. Un cas d'utilisation possible est l'appel d'un *descripteur* d'une classe parente ou sœur.

Note that `super()` is implemented as part of the binding process for explicit dotted attribute lookups such as `super().__getitem__(name)`. It does so by implementing its own `__getattribute__()` method for searching classes in a predictable order that supports cooperative multiple inheritance. Accordingly, `super()` is undefined for implicit lookups using statements or operators such as `super()[name]`.

Notez aussi que, en dehors de sa forme sans argument, `super()` peut être utilisée en dehors des méthodes. La forme à deux arguments est précise et contient tous les arguments nécessaires, donnant les références appropriées. La forme sans argument fonctionne seulement à l'intérieur d'une définition de classe, puisque c'est le compilateur qui donne les détails nécessaires à propos de la classe en cours de définition, ainsi qu'accéder à l'instance courante pour les méthodes ordinaires.

Pour des suggestions pratiques sur la conception de classes coopératives utilisant `super()`, consultez guide to using super().

Modifié dans la version 3.14 : `super` objects are now `pickleable` and `copyable`.

class tuple

class tuple(*iterable*)

Ce n'est pas une fonction, `tuple` est en fait un type de séquence immuable, comme documenté dans *N-uplets*

et *Types séquentiels — list, tuple, range*.

class type(*object*)

class type(*name*, *bases*, *dict*, ***kwds*)

With one argument, return the type of an *object*. The return value is a type object and generally the same object as returned by `object.__class__`.

La fonction native `isinstance()` est recommandée pour tester le type d'un objet, car elle prend en compte l'héritage.

With three arguments, return a new type object. This is essentially a dynamic form of the `class` statement. The *name* string is the class name and becomes the `__name__` attribute. The *bases* tuple contains the base classes and becomes the `__bases__` attribute ; if empty, `object`, the ultimate base of all classes, is added. The *dict* dictionary contains attribute and method definitions for the class body ; it may be copied or wrapped before becoming the `__dict__` attribute. The following two statements create identical `type` objects :

```
>>> class X:
...     a = 1
...
>>> X = type('X', (), dict(a=1))
```

See also :
— Documentation on attributes and methods on classes.
— *Objets type*
Les arguments nommés fournis à la forme à trois arguments sont passés au mécanisme de métaclasse approprié (généralement `__init_subclass__()`) de la même manière que les arguments nommés dans une définition de classe (en plus de *metaclass*).

Voir aussi class-customization.

Modifié dans la version 3.6 : Subclasses of `type` which don't override `type.__new__` may no longer use the one-argument form to get the type of an object.

vars()

vars(*object*)

Return the `__dict__` attribute for a module, class, instance, or any other object with a `__dict__` attribute.

Objects such as modules and instances have an updateable `__dict__` attribute ; however, other objects may have write restrictions on their `__dict__` attributes (for example, classes use a `types.MappingProxyType` to prevent direct dictionary updates).

Without an argument, `vars()` acts like `locals()`.

Une exception `TypeError` est levée si un objet est spécifié mais qu'il n'a pas d'attribut `__dict__` (par exemple, si sa classe définit l'attribut `__slots__`).

Modifié dans la version 3.13 : The result of calling this function without an argument has been updated as described for the `locals()` builtin.

zip(**iterables*, *strict=False*)

Itère sur plusieurs itérables en parallèle, produisant des *n*-uplets avec un élément provenant de chacun.

Exemple :

```
>>> for item in zip([1, 2, 3], ['sugar', 'spice', 'everything nice']):
...     print(item)
...
(1, 'sugar')
(2, 'spice')
(3, 'everything nice')
```

Plus formellement : `zip()` renvoie un itérateur de *n*-uplets, où le i^e *n*-uplet contient le i^e élément de chacun des itérables passés en arguments.

Une autre façon de voir *zip()* est qu'elle transforme les lignes en colonnes et les colonnes en lignes, fournissant la matrice transposée.

zip() est paresseuse : les éléments ne sont traités qu'au moment où l'on itère, par exemple avec une boucle `for` ou en les plaçant dans une *liste*.

Il faut savoir que les itérables passés à *zip()* peuvent avoir des longueurs différentes ; parfois par conception, parfois à cause d'un bogue dans le code qui a préparé ces itérables. Python propose trois approches différentes pour traiter ce problème :

— par défaut, *zip()* s'arrête lorsque l'itérable le plus court est épuisé. Elle ignore les éléments restants dans les itérables plus longs, coupant le résultat à la longueur de l'itérable le plus court :

```
>>> list(zip(range(3), ['fee', 'fi', 'fo', 'fum']))
[(0, 'fee'), (1, 'fi'), (2, 'fo')]
```

— *zip()* est souvent utilisée dans les cas où les itérables sont supposés être de même longueur. Dans ce cas, il est recommandé d'utiliser l'option `strict=True`. Cela produit la même chose que la *zip()* habituelle :

```
>>> list(zip(('a', 'b', 'c'), (1, 2, 3), strict=True))
[('a', 1), ('b', 2), ('c', 3)]
```

Mais, contrairement au comportement par défaut, elle lève une *ValueError* si un itérable est épuisé avant les autres :

```
>>> for item in zip(range(3), ['fee', 'fi', 'fo', 'fum'], strict=True):
...     print(item)
...
(0, 'fee')
(1, 'fi')
(2, 'fo')
Traceback (most recent call last):
  ...
ValueError: zip() argument 2 is longer than argument 1
```

Sans l'argument `strict=True`, tout bogue entraînant des itérables de longueurs différentes est réduit au silence, se manifestant éventuellement comme un bogue difficile à trouver dans une autre partie du programme.

— les itérables plus courts peuvent être remplis avec une valeur constante pour que tous les itérables aient la même longueur. C'est le cas pour *itertools.zip_longest()*.

Cas extrêmes : avec un seul argument itérable, *zip()* renvoie un itérateur de « 1-uplet ». Sans argument, elle renvoie un itérateur vide.

Trucs et astuces :

— il est garanti que les itérables sont évalués de gauche à droite. Cela rend possible de grouper une séquence de données en groupes de taille *n* via `zip(*[iter(s)]*n, strict=True)`. Cela duplique le *même* itérateur `n` fois ; par conséquent le *n*-uplet obtenu contient le résultat de `n` appels à l'itérateur. Cela a pour effet de diviser la séquence en morceaux de taille *n*.

— *zip()* peut être utilisée conjointement avec l'opérateur `*` pour dézipper une liste :

```
>>> x = [1, 2, 3]
>>> y = [4, 5, 6]
>>> list(zip(x, y))
[(1, 4), (2, 5), (3, 6)]
>>> x2, y2 = zip(*zip(x, y))
>>> x == list(x2) and y == list(y2)
True
```

Modifié dans la version 3.10 : ajout de l'argument `strict`.

__import__(*name*, *globals=None*, *locals=None*, *fromlist=()*, *level=0*)

> **Note**
>
> c'est une fonction avancée qui n'est pas fréquemment nécessaire, contrairement à *importlib.import_module()*.

Cette fonction est invoquée via l'instruction `import`. Elle peut être remplacée (en important le module *builtins* et en y remplaçant `builtins.__import__`) afin de changer la sémantique de l'instruction `import`, mais c'est extrêmement déconseillé, car il est plus simple d'utiliser des points d'entrée pour les importations (*import hooks*, voir la **PEP 302**) pour le même résultat sans perturber du code s'attendant à trouver l'implémentation par défaut. L'usage direct de *__import__()* est aussi déconseillé en faveur de *importlib.import_module()*.

La fonction importe le module *name*, utilisant potentiellement *globals* et *locals* pour déterminer comment interpréter le nom dans le contexte d'un paquet. *fromlist* donne le nom des objets ou sous-modules qui devraient être importés du module *name*. L'implémentation standard n'utilise pas l'argument *locals* et n'utilise *globals* que pour déterminer le contexte du paquet de l'instruction `import`.

level permet de choisir entre importation absolue ou relative. `0` (par défaut) force à effectuer uniquement des importations absolues. Une valeur positive indique le nombre de dossiers parents relativement au dossier du module appelant *__import__()* (voir la **PEP 328**).

Lorsque la variable *name* est de la forme `package.module`, normalement, le paquet de plus haut niveau (le nom jusqu'au premier point) est renvoyé, et *pas* le module nommé par *name*. Cependant, lorsqu'un argument *fromlist* est fourni, le module nommé par *name* est renvoyé.

Par exemple, l'instruction `import spam` renvoie un code intermédiaire (*bytecode* en anglais) ressemblant au code suivant :

```
spam = __import__('spam', globals(), locals(), [], 0)
```

L'instruction `import spam.ham` appelle :

```
spam = __import__('spam.ham', globals(), locals(), [], 0)
```

Notez comment *__import__()* renvoie ici le module de plus haut niveau parce que c'est l'objet lié à un nom par l'instruction `import`.

En revanche, l'instruction `from spam.ham import eggs, sausage as saus` donne

```
_temp = __import__('spam.ham', globals(), locals(), ['eggs', 'sausage'], 0)
eggs = _temp.eggs
saus = _temp.sausage
```

Ici le module `spam.ham` est renvoyé par *__import__()*. De cet objet, les noms à importer sont récupérés et assignés à leurs noms respectifs.

Si vous voulez simplement importer un module (potentiellement dans un paquet) par son nom, utilisez *importlib.import_module()*.

Modifié dans la version 3.3 : les valeurs négatives pour *level* ne sont plus prises en charge (et sa valeur par défaut est 0).

Modifié dans la version 3.9 : Quand les options `-E` ou `-I` sont précisées dans la ligne de commande, la variable d'environnement `PYTHONCASEOK` est ignorée.

Notes

CHAPITRE 3

Constantes natives

Un petit nombre de constantes existent dans le *namespace* natif. Elles sont :

`False`

La valeur fausse du type `bool`. Les assignations à `False` ne sont pas autorisées et lèvent une `SyntaxError`.

`True`

La valeur vraie du type `bool`. Les assignations à `True` ne sont pas autorisées et lèvent une `SyntaxError`.

`None`

An object frequently used to represent the absence of a value, as when default arguments are not passed to a function. Assignments to `None` are illegal and raise a `SyntaxError`. `None` is the sole instance of the `NoneType` type.

`NotImplemented`

A special value which should be returned by the binary special methods (e.g. `__eq__()`, `__lt__()`, `__add__()`, `__rsub__()`, etc.) to indicate that the operation is not implemented with respect to the other type ; may be returned by the in-place binary special methods (e.g. `__imul__()`, `__iand__()`, etc.) for the same purpose. It should not be evaluated in a boolean context. `NotImplemented` is the sole instance of the `types.NotImplementedType` type.

> **Note**
>
> When a binary (or in-place) method returns `NotImplemented` the interpreter will try the reflected operation on the other type (or some other fallback, depending on the operator). If all attempts return `NotImplemented`, the interpreter will raise an appropriate exception. Incorrectly returning `NotImplemented` will result in a misleading error message or the `NotImplemented` value being returned to Python code.
>
> Voir *Implémentation des opérations arithmétiques* pour des exemples.

> **Note**
>
> `NotImplementedError` and `NotImplemented` are not interchangeable, even though they have similar names and purposes. See `NotImplementedError` for details on when to use it.

Modifié dans la version 3.9 : Evaluating `NotImplemented` in a boolean context was deprecated.

Modifié dans la version 3.14 : Evaluating `NotImplemented` in a boolean context now raises a `TypeError`. It previously evaluated to `True` and emitted a `DeprecationWarning` since Python 3.9.

Ellipsis

Identique au littéral *points de suspension* (". . ."). Valeur spéciale utilisée principalement de manière conjointe avec la syntaxe de découpage (*slicing*) étendu pour les conteneurs personnalisés. `Ellipsis` est la seule instance de `types.EllipsisType`.

__debug__

Cette constante est vraie si Python n'a pas été démarré avec une option `-O`. Voir aussi l'expression `assert`.

> **Note**
>
> Les noms `None`, `False`, `True` et `__debug__` ne peuvent pas être réassignés (des assignations à ces noms, ou aux noms de leurs attributs, lèvent une `SyntaxError`), donc ils peuvent être considérés comme des "vraies" constantes.

3.1 Constantes ajoutées par le module `site`

Le module `site` (qui est importé automatiquement au démarrage, sauf si l'option de ligne de commande `-S` est donnée ajoute un certain nombre de constantes au *namespace* natif. Elles sont utiles pour l'interpréteur interactif et ne devraient pas être utilisées par des programmes.

quit(*code=None*)

exit(*code=None*)

Objets qui, lorsqu'ils sont représentés, affichent un message comme *"Use quit() or Ctrl-D (i.e. EOF) to exit"*, et lorsqu'ils sont appelés, lèvent un `SystemExit` avec le code de retour spécifié.

help

Object that when printed, prints the message "Type help() for interactive help, or help(object) for help about object.", and when called, acts as described `elsewhere`.

copyright

credits

Objets qui, lorsqu'ils sont affichés ou appelés, affichent le copyright ou les crédits, respectivement.

license

Objet qui, lorsqu'il est affiché, affiche un message comme *"Type license() to see the full license text"*, et lorsqu'il est appelé, affiche le texte complet de la licence dans un style paginé (un écran à la fois).

CHAPITRE 4

Types natifs

Les sections suivantes décrivent les types standards intégrés à l'interpréteur.

Les principaux types natifs sont les numériques, les séquences, les dictionnaires, les classes, les instances et les exceptions.

Certaines classes de collection sont mutables. Les méthodes qui ajoutent, retirent, ou réorganisent leurs éléments sur place, et qui ne renvoient pas un élément spécifique, ne renvoient jamais l'instance de la collection elle-même, mais `None`.

Certaines opérations sont prises en charge par plusieurs types d'objets ; en particulier, pratiquement tous les objets peuvent être comparés en égalité, testés en véridicité (valeur booléenne) et convertis en une chaîne de caractères (avec la fonction `repr()` ou la fonction légèrement différente `str()`). Cette dernière est implicitement utilisée quand un objet est affiché par la fonction `print()`.

4.1 Valeurs booléennes

Tout objet peut être comparé à une valeur booléenne, typiquement dans une condition `if` ou `while` ou comme opérande des opérations booléennes ci-dessous.

Par défaut, tout objet est considéré vrai à moins que sa classe définisse soit une méthode `__bool__()` renvoyant `False` soit une méthode `__len__()` renvoyant zéro lorsqu'elle est appelée avec l'objet. [1] Voici la majorité des objets natifs considérés comme étant faux :

- les constantes définies comme étant fausses : `None` et `False` ;
- zéro de tout type numérique : `0`, `0.0`, `0j`, `Decimal(0)`, `Fraction(0, 1)`
- les chaînes et collections vides : `''`, `()`, `[]`, `{}`, `set()`, `range(0)`

Les opérations et fonctions natives dont le résultat est booléen renvoient toujours `0` ou `False` pour faux et `1` ou `True` pour vrai, sauf indication contraire (exception importante : les opérations booléennes `or` et `and` renvoient toujours l'une de leurs opérandes).

4.2 Opérations booléennes — `and`, `or`, `not`

Ce sont les opérations booléennes, classées par priorité ascendante :

1. Plus d'informations sur ces méthodes spéciales peuvent être trouvées dans le *Python Reference Manual* (customization).

Opération	Résultat	Notes
`x or y`	si *x* est vrai, alors *x*, sinon *y*	(1)
`x and y`	si *x* est faux, alors *x*, sinon *y*	(2)
`not x`	si *x* est faux, alors `True`, sinon `False`	(3)

Notes :

(1) C'est un opérateur court-circuit : il n'évalue le deuxième argument que si le premier est faux.

(2) C'est un opérateur court-circuit : il n'évalue le deuxième argument que si le premier est vrai.

(3) `not` a une priorité inférieure à celle des opérateurs non-booléens, donc `not a == b` est interprété comme `not (a == b)` et `a == not b` est une erreur de syntaxe.

4.3 Comparaisons

Il y a huit opérations de comparaison en Python. Elles ont toutes la même priorité (qui est supérieure à celle des opérations booléennes). Les comparaisons peuvent être enchaînées arbitrairement ; par exemple, `x < y <= z` est équivalent à `x < y and y <= z`, sauf que *y* n'est évalué qu'une seule fois (mais dans les deux cas *z* n'est pas évalué du tout quand `x < y` est faux).

Ce tableau résume les opérations de comparaison :

Opération	Signification
`<`	strictement inférieur
`<=`	inférieur ou égal
`>`	strictement supérieur
`>=`	supérieur ou égal
`==`	égal
`!=`	différent
`is`	identité d'objet
`is not`	contraire de l'identité d'objet

Vous ne pouvez pas tester l'égalité d'objets de types différents, à l'exception des types numériques entre eux. L'opérateur `==` est toujours défini mais pour certains types d'objets (par exemple, les objets de type classe), il est équivalent à `is`. Les opérateurs `<`, `<=`, `>` et `>=` sont définis seulement quand ils ont un sens. Par exemple, ils lèvent une exception `TypeError` lorsque l'un des arguments est un nombre complexe.

Des instances différentes d'une classe sont normalement considérées différentes à moins que la classe ne définisse la méthode `__eq__()`.

Les instances d'une classe ne peuvent pas être ordonnées par rapport à d'autres instances de la même classe, ou d'autres types d'objets, à moins que la classe ne définisse suffisamment de méthodes parmi `__lt__()`, `__le__()`, `__gt__()` et `__ge__()` (en général, `__lt__()` et `__eq__()` sont suffisantes, si vous voulez les significations classiques des opérateurs de comparaison).

Le comportement des opérateurs `is` et `is not` ne peut pas être personnalisé ; aussi ils peuvent être appliqués à deux objets quelconques et ne lèvent jamais d'exception.

Deux autres opérations avec la même priorité syntaxique, `in` et `not in`, sont pris en charge par les types *iterable* ou qui implémentent la méthode `__contains__()`.

4.4 Types numériques — `int`, `float`, `complex`

Il existe trois types numériques distincts : les entiers (*integers*), les nombres flottants (*floating point numbers*) et les nombres complexes (*complex numbers*). En outre, les booléens sont un sous-type des entiers. Les entiers ont une précision illimitée. Les nombres à virgule flottante sont généralement implémentés en utilisant des `double` en C ; des informations sur la précision et la représentation interne des nombres à virgule flottante pour la machine sur laquelle

le programme est en cours d'exécution sont disponibles dans `sys.float_info`. Les nombres complexes ont une partie réelle et une partie imaginaire, qui sont chacune des nombres à virgule flottante. Pour extraire ces parties d'un nombre complexe *z*, utilisez `z.real` et `z.imag`. (La bibliothèque standard comprend les types numériques additionnels `fractions.Fraction` pour les rationnels et `decimal.Decimal` pour les nombres à virgule flottante avec une précision définissable par l'utilisateur.)

Les nombres sont créés par des littéraux numériques ou sont le résultat de fonctions natives ou d'opérateurs. Les entiers littéraux basiques (y compris leur forme hexadécimale, octale et binaire) donnent des entiers. Les nombres littéraux contenant un point décimal (NdT : notation anglo-saxonne de la virgule) ou un exposant donnent des nombres à virgule flottante. Suffixer `'j'` ou `'J'` à un nombre littéral donne un nombre imaginaire (un nombre complexe avec une partie réelle nulle) que vous pouvez ajouter à un nombre (entier ou à virgule flottante) pour obtenir un nombre complexe avec une partie réelle et une partie imaginaire.

Les constructeurs `int()`, `float()` et `complex()` peuvent être utilisés pour produire des nombres d'un type numérique spécifique.

Python fully supports mixed arithmetic : when a binary arithmetic operator has operands of different numeric types, the operand with the "narrower" type is widened to that of the other, where integer is narrower than floating point. Arithmetic with complex and real operands is defined by the usual mathematical formula, for example :

```
x + complex(u, v) = complex(x + u, v)
x * complex(u, v) = complex(x * u, x * v)
```

A comparison between numbers of different types behaves as though the exact values of those numbers were being compared. [2]

Tous les types numériques (sauf complexe) gèrent les opérations suivantes (pour les priorités des opérations, voir operator-summary) :

Opération	Résultat	Notes	Documentation complète
`x + y`	somme de *x* et *y*		
`x - y`	différence de *x* et *y*		
`x * y`	produit de *x* et *y*		
`x / y`	quotient de *x* et *y*		
`x // y`	quotient entier de *x* et *y*	(1)(2)	
`x % y`	reste de `x / y`	(2)	
`-x`	négatif de *x*		
`+x`	*x* inchangé		
`abs(x)`	valeur absolue de *x*		`abs()`
`int(x)`	*x* converti en nombre entier	(3)(6)	`int()`
`float(x)`	*x* converti en nombre à virgule flottante	(4)(6)	`float()`
`complex(re, im)`	un nombre complexe avec *re* pour partie réelle et *im* pour partie imaginaire. *im* vaut zéro par défaut.	(6)	`complex()`
`c.conjugate()`	conjugué du nombre complexe *c*		
`divmod(x, y)`	la paire `(x // y, x % y)`	(2)	`divmod()`
`pow(x, y)`	*x* à la puissance *y*	(5)	`pow()`
`x ** y`	*x* à la puissance *y*	(5)	

Notes :

(1) Également appelé division entière. Pour les opérandes de type `int`, la valeur résultante est de type `int`. Pour les opérandes de type `float`,la valeur résultante est de type `float`. En général, la valeur résultante est strictement entière, bien que le type du résultat ne soit pas nécessairement *int*. Le résultat est toujours arrondi vers moins l'infini : `1//2` vaut `0`, `(-1)//2` vaut `-1`, `1//(-2)` vaut `-1`, et `(-1)//(-2)` vaut `0`.

2. Par conséquent, la liste `[1, 2]` est considérée égale à `[1.0, 2.0]`. Idem avec des *n*-uplets.

(2) Pas pour les nombres complexes. Convertissez-les plutôt en nombres flottants à l'aide de `abs()` si c'est approprié.
(3) La conversion de `float` en `int` tronque la partie fractionnaire. Voir les fonctions `math.floor()` et `math.ceil()` pour d'autres conversions.
(4) *float* accepte aussi les chaînes *nan* et *inf* avec un préfixe optionnel + ou − pour *Not a Number* (*NaN*) et les infinis positif ou négatif.
(5) Python définit `pow(0, 0)` et `0 ** 0` valant `1`, puisque c'est courant pour les langages de programmation, et logique.
(6) Les littéraux numériques acceptés comprennent les chiffres `0` à `9` ou tout équivalent Unicode (caractères avec la propriété `Nd`).

See the Unicode Standard for a complete list of code points with the `Nd` property.

Tous types `numbers.Real` (`int` et `float`) comprennent également les opérations suivantes :

Opération	Résultat
`math.trunc(x)`	*x* tronqué à l'`Integral`
`round(x[, n])`	*x* arrondi à *n* chiffres, arrondissant la moitié au pair. Si *n* est omis, la valeur par défaut est 0.
`math.floor(x)`	le plus grand `Integral` ≤ *x*
`math.ceil(x)`	le plus petit `Integral` ≥ *x*

Pour d'autres opérations numériques voir les modules `math` et `cmath`.

4.4.1 Opérations sur les bits des nombres entiers

Les opérations bit à bit n'ont de sens que pour les entiers relatifs. Le résultat d'une opération bit à bit est calculé comme si elle était effectuée en complément à deux avec un nombre infini de bits de signe.

Les priorités de toutes les opérations bit à bit à deux opérandes sont inférieures à celles des opérations numériques et plus élevées que les comparaisons ; l'opération unaire ~ a la même priorité que les autres opérations numériques unaires (+ et −).

Ce tableau répertorie les opérations binaires triées par priorité ascendante :

Opération	Résultat	Notes
`x | y`	*OU* bit à bit de *x* et *y*	(4)
`x ^ y`	*OU exclusif* bit à bit de *x* et *y*	(4)
`x & y`	*ET* bit à bit de *x* et *y*	(4)
`x << n`	*x* décalé vers la gauche de *n* bits	(1)(2)
`x >> n`	*x* décalé vers la droite de *n* bits	(1)(3)
`~x`	les bits de *x*, inversés	

Notes :

(1) Des valeurs de décalage négatives sont illégales et provoquent une exception `ValueError`.
(2) Un décalage à gauche de *n* bits est équivalent à la multiplication par `pow(2, n)`.
(3) Un décalage à droite de *n* les bits est équivalent à la division par `pow(2, n)`.
(4) Effectuer ces calculs avec au moins un bit d'extension de signe supplémentaire dans une représentation finie du complément à deux éléments (une largeur de bit fonctionnelle de `1 + max(x.bit_length(), y.bit_length()` ou plus) est suffisante pour obtenir le même résultat que s'il y avait un nombre infini de bits de signe.

4.4.2 Méthodes supplémentaires sur les entiers

Le type *int* implémente la *classe mère abstraite* `numbers.Integral`. Il fournit aussi quelques autres méthodes :

`int.bit_length()`

Renvoie le nombre de bits nécessaires pour représenter un nombre entier en binaire, à l'exclusion du signe et des zéros non significatifs :

```
>>> n = -37
>>> bin(n)
'-0b100101'
>>> n.bit_length()
6
```

Plus précisément, si `x` est différent de zéro, `x.bit_length()` est le nombre entier positif unique, `k` tel que `2**(k-1) <= abs(x) < 2**k`. Équivalemment, quand `abs(x)` est assez petit pour avoir un logarithme correctement arrondi, `k = 1 + int(log(abs(x), 2))`. Si `x` est nul, alors `x.bit_length()` donne `0`.

Équivalent à :

```
def bit_length(self):
    s = bin(self)       # binary representation:  bin(-37) --> '-0b100101'
    s = s.lstrip('-0b') # remove leading zeros and minus sign
    return len(s)       # len('100101') --> 6
```

Ajouté dans la version 3.1.

`int.bit_count()`

Renvoie le nombre de 1 dans la représentation binaire de la valeur absolue de l'entier. On la connait également sous le nom de dénombrement de la population. Par exemple :

```
>>> n = 19
>>> bin(n)
'0b10011'
>>> n.bit_count()
3
>>> (-n).bit_count()
3
```

Équivalent à :

```
def bit_count(self):
    return bin(self).count("1")
```

Ajouté dans la version 3.10.

`int.to_bytes`(*length=1*, *byteorder='big'*, **, signed=False*)

Renvoie un tableau d'octets représentant un nombre entier.

```
>>> (1024).to_bytes(2, byteorder='big')
b'\x04\x00'
>>> (1024).to_bytes(10, byteorder='big')
b'\x00\x00\x00\x00\x00\x00\x00\x00\x04\x00'
>>> (-1024).to_bytes(10, byteorder='big', signed=True)
b'\xff\xff\xff\xff\xff\xff\xff\xff\xfc\x00'
>>> x = 1000
>>> x.to_bytes((x.bit_length() + 7) // 8, byteorder='little')
b'\xe8\x03'
```

L'entier est représenté en utilisant *length* octets, dont la valeur par défaut est 1. Une exception `OverflowError` est levée s'il n'est pas possible de représenter l'entier avec le nombre donné d'octets.

L'argument *byteorder* détermine l'ordre des octets utilisé pour représenter le nombre entier, la valeur par défaut étant `"big"`. Si *byteorder* est `"big"`, l'octet le plus significatif est au début du tableau d'octets. Si *byteorder* est `"little"`, l'octet le plus significatif est à la fin du tableau d'octets.

L'argument *signed* détermine si le complément à deux est utilisé pour représenter le nombre entier. Si *signed* est `False` et qu'un entier négatif est donné, une exception `OverflowError` est levée. La valeur par défaut pour *signed* est `False`.

Les valeurs par défaut peuvent être utilisées pour transformer facilement un entier en un objet à un seul octet :

```
>>> (65).to_bytes()
b'A'
```

Cependant, lorsque vous utilisez les arguments par défaut, n'essayez pas de convertir une valeur supérieure à 255 ou vous lèverez une `OverflowError` :

Équivalent à :

```
def to_bytes(n, length=1, byteorder='big', signed=False):
    if byteorder == 'little':
        order = range(length)
    elif byteorder == 'big':
        order = reversed(range(length))
    else:
        raise ValueError("byteorder must be either 'little' or 'big'")

    return bytes((n >> i*8) & 0xff for i in order)
```

Ajouté dans la version 3.2.

Modifié dans la version 3.11 : ajout de valeurs par défaut pour les arguments `length` et `byteorder`.

classmethod `int.`**`from_bytes`**(*bytes*, *byteorder='big'*, **, signed=False*)

Renvoie le nombre entier représenté par le tableau d'octets fourni.

```
>>> int.from_bytes(b'\x00\x10', byteorder='big')
16
>>> int.from_bytes(b'\x00\x10', byteorder='little')
4096
>>> int.from_bytes(b'\xfc\x00', byteorder='big', signed=True)
-1024
>>> int.from_bytes(b'\xfc\x00', byteorder='big', signed=False)
64512
>>> int.from_bytes([255, 0, 0], byteorder='big')
16711680
```

L'argument *bytes* doit être soit un *objet octet-compatible*, soit un itérable produisant des *bytes*.

L'argument *byteorder* détermine l'ordre des octets utilisé pour représenter le nombre entier, la valeur par défaut étant `"big"`. Si *byteorder* est `"big"`, l'octet le plus significatif est au début du tableau d'octets. Si *byteorder* est `"little"`, l'octet le plus significatif est à la fin du tableau d'octets. Pour demander l'ordre natif des octets du système hôte, donnez `sys.byteorder` comme *byteorder*.

L'argument *signed* indique si le complément à deux est utilisé pour représenter le nombre entier.

Équivalent à :

```
def from_bytes(bytes, byteorder='big', signed=False):
    if byteorder == 'little':
        little_ordered = list(bytes)
    elif byteorder == 'big':
        little_ordered = list(reversed(bytes))
```

(suite sur la page suivante)

(suite de la page précédente)

```
    else:
        raise ValueError("byteorder must be either 'little' or 'big'")

    n = sum(b << i*8 for i, b in enumerate(little_ordered))
    if signed and little_ordered and (little_ordered[-1] & 0x80):
        n -= 1 << 8*len(little_ordered)

    return n
```

Ajouté dans la version 3.2.

Modifié dans la version 3.11 : ajout de la valeur par défaut pour l'argument `byteorder`.

`int.`**`as_integer_ratio`**`()`

Return a pair of integers whose ratio is equal to the original integer and has a positive denominator. The integer ratio of integers (whole numbers) is always the integer as the numerator and `1` as the denominator.

Ajouté dans la version 3.8.

`int.`**`is_integer`**`()`

Returns `True`. Exists for duck type compatibility with *`float.is_integer()`*.

Ajouté dans la version 3.12.

4.4.3 Méthodes supplémentaires sur les nombres à virgule flottante

Le type *float* implémente la *classe mère abstraite* `numbers.Real` et a également les méthodes suivantes.

classmethod `float.`**`from_number`**`(`*x*`)`

Class method to return a floating-point number constructed from a number *x*.

If the argument is an integer or a floating-point number, a floating-point number with the same value (within Python's floating-point precision) is returned. If the argument is outside the range of a Python float, an `OverflowError` will be raised.

For a general Python object `x`, `float.from_number(x)` delegates to `x.__float__()`. If `__float__()` is not defined then it falls back to `__index__()`.

Ajouté dans la version 3.14.

`float.`**`as_integer_ratio`**`()`

Return a pair of integers whose ratio is exactly equal to the original float. The ratio is in lowest terms and has a positive denominator. Raises `OverflowError` on infinities and a `ValueError` on NaNs.

`float.`**`is_integer`**`()`

Renvoie `True` si l'instance de *float* est finie avec une valeur entière, et `False` autrement :

```
>>> (-2.0).is_integer()
True
>>> (3.2).is_integer()
False
```

Deux méthodes prennent en charge la conversion vers et à partir de chaînes hexadécimales. Étant donné que les *float* de Python sont stockés en interne sous forme de nombres binaires, la conversion d'un *float* depuis ou vers une chaine décimale implique généralement une petite erreur d'arrondi. En revanche, les chaînes hexadécimales permettent de représenter exactement les nombres à virgule flottante. Cela peut être utile lors du débogage, et dans un travail numérique.

`float.`**`hex`**`()`

Renvoie une représentation d'un nombre à virgule flottante sous forme de chaîne hexadécimale. Pour les nombres à virgule flottante finis, cette représentation comprendra toujours un préfixe `0x`, un suffixe `p` et un exposant.

classmethod float.**fromhex**(*s*)

Méthode de classe pour obtenir le *float* représenté par une chaîne de caractères hexadécimale *s*. La chaîne *s* peut contenir des espaces avant et après le nombre.

Notez que `float.hex()` est une méthode d'instance, alors que `float.fromhex()` est une méthode de classe.

Une chaîne hexadécimale prend la forme :

```
[sign] ['0x'] integer ['.' fraction] ['p' exponent]
```

où `sign` peut être soit `+` soit `-`, `integer` et `fraction` sont des chaînes de chiffres hexadécimaux, et `exponent` est un entier décimal facultativement signé. La casse n'est pas significative, et il doit y avoir au moins un chiffre hexadécimal soit dans le nombre entier soit dans la fraction. Cette syntaxe est similaire à la syntaxe spécifiée dans la section 6.4.4.2 de la norme C99, et est aussi la syntaxe utilisée à partir de Java 1.5. En particulier, la sortie de `float.hex()` est utilisable comme valeur hexadécimale à virgule flottante littérale en C ou Java, et des chaînes hexadécimales produites en C via un format `%a` ou Java via `Double.toHexString` sont acceptées par `float.fromhex()`.

Notez que l'exposant est écrit en décimal plutôt qu'en hexadécimal, et qu'il donne la puissance de 2 par lequel multiplier le coefficient. Par exemple, la chaîne hexadécimale `0x3.a7p10` représente le nombre à virgule flottante `(3 + 10./16 + 7./16**2) *2.0**10`, ou `3740.0` :

```
>>> float.fromhex('0x3.a7p10')
3740.0
```

L'application de la conversion inverse à `3740.0` donne une chaîne hexadécimale différente représentant le même nombre :

```
>>> float.hex(3740.0)
'0x1.d380000000000p+11'
```

4.4.4 Additional Methods on Complex

The `complex` type implements the `numbers.Complex` *abstract base class*. `complex` also has the following additional methods.

classmethod complex.**from_number**(*x*)

Class method to convert a number to a complex number.

For a general Python object `x`, `complex.from_number(x)` delegates to `x.__complex__()`. If `__complex__()` is not defined then it falls back to `__float__()`. If `__float__()` is not defined then it falls back to `__index__()`.

Ajouté dans la version 3.14.

4.4.5 Hachage des types numériques

For numbers `x` and `y`, possibly of different types, it's a requirement that `hash(x) == hash(y)` whenever `x == y` (see the `__hash__()` method documentation for more details). For ease of implementation and efficiency across a variety of numeric types (including `int`, `float`, `decimal.Decimal` and `fractions.Fraction`) Python's hash for numeric types is based on a single mathematical function that's defined for any rational number, and hence applies to all instances of `int` and `fractions.Fraction`, and all finite instances of `float` and `decimal.Decimal`. Essentially, this function is given by reduction modulo `P` for a fixed prime `P`. The value of `P` is made available to Python as the `modulus` attribute of `sys.hash_info`.

actuellement, le premier utilisé est `P = 2 ** 31 - 1` sur des machines dont les *longs* en C sont de 32 bits `P = 2 ** 61 - 1` sur des machines dont les *longs* en C font 64 bits.

Voici les règles en détail :

— Si `x = m / n` est un nombre rationnel non négatif et `n` n'est pas divisible par `P`, définir `hash(x)` comme `m * invmod(n, P) % P`, où `invmod(n, P)` donne l'inverse de `n` modulo `P`.

— Si `x = m / n` est un nombre rationnel non négatif et `n` est divisible par `P` (mais `m` ne l'est pas), alors `n` n'a pas de modulo inverse `P` et la règle ci-dessus n'est pas applicable ; dans ce cas définir `hash(x)` comme étant la valeur de la constante `sys.hash_info.inf`.
— Si `x = m / n` est un nombre rationnel négatif définir `hash(x)` comme `-hash(-x)`. Si le résultat est `-1`, le remplacer par `-2`.
— Les valeurs particulières `sys.hash_info.inf` et `-sys.hash_info.inf` sont utilisées comme valeurs de hachage pour l'infini positif et l'infini négatif (respectivement).
— Pour un nombre *complexe* `z`, les valeurs de hachage des parties réelles et imaginaires sont combinées en calculant `hash(z.real) + sys.hash_info.imag * hash(z.imag)`, réduit au modulo `2**sys.hash_info.width` de sorte qu'il se trouve dans `range(-2**(sys.hash_info.width - 1), 2**(sys.hash_info.width - 1))`. Encore une fois, si le résultat est `-1`, il est remplacé par `-2`.

Afin de clarifier les règles ci-dessus, voici quelques exemples de code Python, équivalent à la fonction de hachage native, pour calculer le hachage d'un nombre rationnel, d'un *float*, ou d'un *complex* :

```
import sys, math

def hash_fraction(m, n):
    """Compute the hash of a rational number m / n.

    Assumes m and n are integers, with n positive.
    Equivalent to hash(fractions.Fraction(m, n)).

    """
    P = sys.hash_info.modulus
    # Remove common factors of P.  (Unnecessary if m and n already coprime.)
    while m % P == n % P == 0:
        m, n = m // P, n // P

    if n % P == 0:
        hash_value = sys.hash_info.inf
    else:
        # Fermat's Little Theorem: pow(n, P-1, P) is 1, so
        # pow(n, P-2, P) gives the inverse of n modulo P.
        hash_value = (abs(m) % P) * pow(n, P - 2, P) % P
    if m < 0:
        hash_value = -hash_value
    if hash_value == -1:
        hash_value = -2
    return hash_value

def hash_float(x):
    """Compute the hash of a float x."""

    if math.isnan(x):
        return object.__hash__(x)
    elif math.isinf(x):
        return sys.hash_info.inf if x > 0 else -sys.hash_info.inf
    else:
        return hash_fraction(*x.as_integer_ratio())

def hash_complex(z):
    """Compute the hash of a complex number z."""

    hash_value = hash_float(z.real) + sys.hash_info.imag * hash_float(z.imag)
    # do a signed reduction modulo 2**sys.hash_info.width
    M = 2**(sys.hash_info.width - 1)
    hash_value = (hash_value & (M - 1)) - (hash_value & M)
    if hash_value == -1:
```

(suite sur la page suivante)

(suite de la page précédente)

```
        hash_value = -2
    return hash_value
```

4.5 Boolean Type - bool

Booleans represent truth values. The *bool* type has exactly two constant instances : `True` and `False`.

The built-in function *bool()* converts any value to a boolean, if the value can be interpreted as a truth value (see section *Valeurs booléennes* above).

For logical operations, use the *boolean operators* `and`, `or` and `not`. When applying the bitwise operators `&`, `|`, `^` to two booleans, they return a bool equivalent to the logical operations "and", "or", "xor". However, the logical operators `and`, `or` and `!=` should be preferred over `&`, `|` and `^`.

Obsolète depuis la version 3.12 : The use of the bitwise inversion operator `~` is deprecated and will raise an error in Python 3.16.

bool is a subclass of *int* (see *Types numériques — int, float, complex*). In many numeric contexts, `False` and `True` behave like the integers 0 and 1, respectively. However, relying on this is discouraged ; explicitly convert using *int()* instead.

4.6 Les types itérateurs

Python gère un concept d'itération sur les conteneurs. Il l'implémente en utilisant deux méthodes distinctes qui permettent aux classes définies par l'utilisateur de devenir itérables. Les séquences, décrites plus bas en détail, savent toujours gérer les méthodes d'itération.

Une méthode doit être définie afin que les objets conteneurs prennent en charge *l'itération* :

`container.__iter__()`

Renvoie un objet *itérateur*. L'objet doit implémenter le protocole d'itération décrit ci-dessous. Si un conteneur prend en charge différents types d'itération, d'autres méthodes peuvent être fournies pour obtenir spécifiquement les itérateurs pour ces types d'itération. (Exemple d'un objet gérant plusieurs formes d'itération : une structure d'arbre pouvant être parcourue en largeur ou en profondeur.) Cette méthode correspond à l'attribut `tp_iter` de la structure du type des objets Python dans l'API Python/C.

Les itérateurs eux-mêmes doivent implémenter les deux méthodes suivantes, qui forment ensemble le *protocole d'itération* :

`iterator.__iter__()`

Renvoie *l'objet itérateur* lui-même. C'est nécessaire pour permettre à la fois à des conteneurs et des itérateurs d'être utilisés avec les instructions `for` et `in`. Cette méthode correspond à l'attribut `tp_iter` de la structure des types des objets Python dans l'API Python/C.

`iterator.__next__()`

Renvoie l'élément suivant de l'*itérateur*. S'il n'y a pas d'autres éléments, une exception *StopIteration* est levée. Cette méthode correspond à l'attribut `tp_iternext` de la structure du type des objets Python dans l'API Python/C.

Python définit plusieurs objets itérateurs pour itérer sur les types standards ou spécifiques de séquence, de dictionnaires et d'autres formes plus spécialisées. Les types spécifiques ne sont pas importants au-delà de leur implémentation du protocole d'itération.

Dès que la méthode *__next__()* lève une exception *StopIteration*, elle doit continuer à le faire lors des appels ultérieurs. Les implémentations qui ne respectent pas cette propriété sont considérées cassées.

4.6.1 Types générateurs

Python's *generator*s provide a convenient way to implement the iterator protocol. If a container object's `__iter__()` method is implemented as a generator, it will automatically return an iterator object (technically, a generator object) supplying the `__iter__()` and `__next__()` methods. More information about generators can be found in the documentation for the yield expression.

4.7 Types séquentiels — `list`, `tuple`, `range`

Il existe trois types séquentiels élémentaires : les listes (objets *list*), *n*-uplets (objets *tuple*) et les intervalles (objets *range*). D'autres types séquentiels spécifiques au traitement de *données binaires* et *chaînes de caractères* sont décrits dans des sections dédiées.

4.7.1 Opérations communes sur les séquences

Les opérations dans le tableau ci-dessous sont prises en charge par la plupart des types séquentiels, variables et immuables. La classe mère abstraite `collections.abc.Sequence` est fournie pour aider à implémenter correctement ces opérations sur les types séquentiels personnalisés.

Ce tableau répertorie les opérations sur les séquences triées par priorité ascendante. Dans le tableau, *s* et *t* sont des séquences du même type, *n*, *i*, *j* et *k* sont des nombres entiers et *x* est un objet arbitraire qui répond à toutes les restrictions de type et de valeur imposée par *s*.

Les opérations `in` et `not in` ont les mêmes priorités que les opérations de comparaison. Les opérations `+` (concaténation) et `*` (répétition) ont la même priorité que les opérations numériques correspondantes [3].

Opération	Résultat	Notes
`x in s`	`True` si un élément de *s* est égal à *x*, sinon `False`	(1)
`x not in s`	`False` si un élément de *s* est égal à *x*, sinon `True`	(1)
`s + t`	la concaténation de *s* et *t*	(6)(7)
`s * n` ou `n * s`	équivalent à ajouter *s* *n* fois à lui-même	(2)(7)
`s[i]`	i^{e} élément de *s* en commençant par 0	(3)
`s[i:j]`	tranche (*slice*) de *s* de *i* à *j*	(3)(4)
`s[i:j:k]`	tranche (*slice*) de *s* de *i* à *j* avec un pas de *k*	(3)(5)
`len(s)`	longueur de *s*	
`min(s)`	plus petit élément de *s*	
`max(s)`	plus grand élément de *s*	
`s.index(x[, i[, j]])`	indice de la première occurrence de *x* dans *s* (à ou après l'indice *i* et avant l'indice *j*)	(8)
`s.count(x)`	nombre total d'occurrences de *x* dans *s*	

Les séquences du même type gèrent également la comparaison. En particulier, les *n*-uplets et les listes sont comparés lexicographiquement en comparant les éléments correspondants. Cela signifie que, pour que deux séquences soit égales, les éléments les constituant doivent être égaux deux à deux et les deux séquences doivent être du même type et de la même longueur. (Pour plus de détails voir comparisons dans la référence du langage.)

Les itérateurs avant et arrière sur des séquences modifiables accèdent aux valeurs à l'aide d'un indice. Cet indice continue à avancer (ou à reculer) même si la séquence sous-jacente est modifiée. L'itérateur ne se termine que lorsqu'une `IndexError` ou une `StopIteration` est rencontrée (ou lorsque l'indice tombe en dessous de zéro).

Notes :

(1) Bien que les opérations `in` et `not in` ne soient généralement utilisées que pour les tests d'appartenance simple, certaines séquences spécialisées (telles que `str`, `bytes` et `bytearray`) les utilisent aussi pour tester l'existence de sous-séquences :

```
>>> "gg" in "eggs"
True
```

3. Nécessairement, puisque l'analyseur ne peut pas discerner le type des opérandes.

(2) Les valeurs de *n* plus petites que `0` sont traitées comme `0` (ce qui donne une séquence vide du même type que *s*). Notez que les éléments de *s* ne sont pas copiés ; ils sont référencés plusieurs fois. Cela hante souvent de nouveaux développeurs Python, typiquement :

```
>>> lists = [[]] * 3
>>> lists
[[], [], []]
>>> lists[0].append(3)
>>> lists
[[3], [3], [3]]
```

Ce qui est arrivé est que `[[]]` est une liste à un élément contenant une liste vide, de sorte que les trois éléments de `[[]] * 3` sont des références à cette seule liste vide. Modifier l'un des éléments de `lists` modifie cette liste unique. Vous pouvez créer une liste des différentes listes de cette façon :

```
>>> lists = [[] for i in range(3)]
>>> lists[0].append(3)
>>> lists[1].append(5)
>>> lists[2].append(7)
>>> lists
[[3], [5], [7]]
```

De plus amples explications sont disponibles dans la FAQ à la question faq-multidimensional-list.

(3) Si *i* ou *j* sont négatifs, l'indice est relatif à la fin de la séquence *s* : `len(s) + i` ou `len(s) + j` est substitué. Mais notez que `-0` est toujours `0`.

(4) La tranche de *s* de *i* à *j* est définie comme la séquence d'éléments d'indices *k* tels que `i <= k < j`. Si *i* ou *j* est supérieur à `len(s)`, `len(s)` est utilisé. Si *i* est omis ou `None`, `0` est utilisé. Si *j* est omis ou `None`, `len(s)` est utilisé. Si *i* est supérieur ou égal à *j*, la tranche est vide.

(5) La tranche de *s* de *i* à *j* avec un pas de *k* est définie comme la séquence d'éléments d'indices `x = i + n*k` tels que `0 <= n < (j-i)/k`. En d'autres termes, les indices sont `i`, `i+k`, `i+2*k`, `i+3*k` et ainsi de suite, en arrêtant lorsque *j* est atteint (mais jamais inclus). Si *k* est positif, *i* et *j* sont réduits, s'ils sont plus grands, à `len(s)`. Si *k* est négatif, *i* et *j* sont réduits à `len(s) - 1` s'ils sont plus grands. Si *i* ou *j* sont omis ou sont `None`, ils deviennent des valeurs « limites » (la limite haute ou basse dépend du signe de *k*). Notez que *k* ne peut pas valoir zéro. Si *k* est `None`, il est traité comme `1`.

(6) Concaténer des séquences immuables donne toujours un nouvel objet. Cela signifie que la construction d'une séquence par concaténations répétées aura une durée d'exécution quadratique par rapport à la longueur de la séquence totale. Pour obtenir un temps d'exécution linéaire, vous devez utiliser l'une des alternatives suivantes :
- si vous concaténez des *`str`*, vous pouvez construire une liste puis utiliser *`str.join()`* à la fin, ou bien écrire dans une instance de *`io.StringIO`* et récupérer sa valeur lorsque vous avez terminé ;
- si vous concaténez des *`bytes`*, vous pouvez aussi utiliser *`bytes.join()`* ou *`io.BytesIO`*, ou vous pouvez faire les concaténations sur place avec un objet *`bytearray`*. Les objets *`bytearray`* sont mutables et ont un mécanisme de sur-allocation efficace ;
- si vous concaténez des *`n-uplets`*, utilisez plutôt *extend* sur une *`list`* ;
- pour les autres types, cherchez dans la documentation de la classe concernée.

(7) Certains types séquentiels (tels que *`range`*) ne gèrent que des séquences qui suivent des modèles spécifiques, et donc ne prennent pas en charge la concaténation ou la répétition.

(8) `index` lève une exception *`ValueError`* quand *x* ne se trouve pas dans *s*. Toutes les implémentations ne gèrent pas les deux paramètres supplémentaires *i* et *j*. Ces deux arguments permettent de chercher efficacement dans une sous-séquence de la séquence. Donner ces arguments est plus ou moins équivalent à `s[i:j].index(x)`, sans copier les données ; l'indice renvoyé est relatif au début de la séquence et non au début de la tranche.

4.7.2 Types de séquences immuables

La seule opération que les types de séquences immuables implémentent et qui n'est pas implémentée par les types de séquences mutables est la fonction native *`hash()`*.

Cette implémentation permet d'utiliser des séquences immuables, comme les instances de *`n-uplets`*, en tant que clés de *`dict`* et stockées dans les instances de *`set`* et *`frozenset`*.

Essayer de hacher une séquence immuable qui contient des valeurs non hachables lève une `TypeError`.

4.7.3 Types de séquences mutables

Les opérations dans le tableau ci-dessous sont définies sur les types de séquences mutables. La classe mère abstraite `collections.abc.MutableSequence` est prévue pour faciliter l'implémentation correcte de ces opérations sur les types de séquences personnalisées.

Dans le tableau ci-dessosus, *s* est une instance d'un type de séquence mutable, *t* est un objet itérable et *x* est un objet arbitraire qui répond à toutes les restrictions de type et de valeur imposées par *s* (par exemple, `bytearray` accepte uniquement des nombres entiers qui répondent à la restriction de la valeur `0 <= x <= 255`).

Opération	Résultat	Notes
`s[i] = x`	l'élément *i* de *s* est remplacé par *x*	
`s[i:j] = t`	la tranche de *s* de *i* à *j* est remplacée par le contenu de l'itérable *t*	
`del s[i:j]`	identique à `s[i:j] = []`	
`s[i:j:k] = t`	les éléments de `s[i:j:k]` sont remplacés par ceux de *t*	(1)
`del s[i:j:k]`	supprime les éléments de `s[i:j:k]` de la liste	
`s.append(x)`	ajoute *x* à la fin de la séquence (identique à `s[len(s):len(s)] = [x]`)	
`s.clear()`	supprime tous les éléments de *s* (identique à `del s[:]`)	(5)
`s.copy()`	crée une copie superficielle de *s* (identique à `s[:]`)	(5)
`s.extend(t)` ou `s += t`	étend *s* avec le contenu de *t* (proche de `s[len(s):len(s)] = t`)	
`s *= n`	met à jour *s* avec son contenu répété *n* fois	(6)
`s.insert(i, x)`	insère *x* dans *s* à l'indice donné par *i* (identique à `s[i:i] = [x]`)	
`s.pop()` ou `s.pop(i)`	récupère l'élément à la position *i* et le supprime de *s*	(2)
`s.remove(x)`	removes the first item from *s* where `s[i]` is equal to *x*	(3)
`s.reverse()`	inverse sur place les éléments de *s*	(4)

Notes :

(1) If *k* is not equal to `1`, *t* must have the same length as the slice it is replacing.

(2) L'argument optionnel *i* vaut `-1` par défaut, afin que, par défaut, le dernier élément soit retiré et renvoyé.

(3) `remove()` lève une exception `ValueError` si *x* ne se trouve pas dans *s*.

(4) La méthode `reverse()` modifie la séquence sur place pour économiser de l'espace lors du traitement de grandes séquences. Pour rappeler aux utilisateurs qu'elle a un effet de bord, elle ne renvoie pas la séquence inversée.

(5) `clear()` et `copy()` sont incluses pour la compatibilité avec les interfaces des conteneurs mutables qui ne gèrent pas les opérations de découpage (comme `dict` et `set`). `copy()` ne fait pas partie des classes mères abstraites (*ABC*) de `collections.abc.MutableSequence`, mais la plupart des classes implémentées gérant des séquences la proposent.

Ajouté dans la version 3.3 : méthodes `clear()` et `copy()`.

(6) La valeur *n* est un entier, ou un objet implémentant `__index__()`. Zéro et les valeurs négatives de *n* permettent d'effacer la séquence. Les éléments dans la séquence ne sont pas copiés ; ils sont référencés plusieurs fois, comme expliqué pour `s * n` dans *Opérations communes sur les séquences*.

4.7.4 Listes

Les listes sont des séquences mutables, généralement utilisées pour stocker des collections d'éléments homogènes (le degré de similitude varie selon l'usage).

class list([*iterable*])

Les listes peuvent être construites de différentes manières :

— en utilisant une paire de crochets pour indiquer une liste vide : `[]` ;
— au moyen de crochets, en séparant les éléments par des virgules : `[a]`, `[a, b, c]` ;
— en utilisant une liste en compréhension : `[x for x in iterable]` ;
— en utilisant le constructeur du type : `list()` ou `list(iterable)`.

Le constructeur crée une liste dont les éléments sont les mêmes et dans le même ordre que les éléments d'*iterable*. *iterable* peut être une séquence, un conteneur qui prend en charge l'itération, ou un itérateur. Si *iterable* est déjà une liste, une copie est faite et renvoyée, comme avec `iterable[:]`. Par exemple, `list('abc')` renvoie `['a', 'b', 'c']` et `list( (1, 2, 3) )` renvoie `[1, 2, 3]`. Si aucun argument est donné, le constructeur crée une nouvelle liste vide, `[]`.

De nombreuses autres opérations produisent des listes, comme la fonction native `sorted()`.

Les listes gèrent toutes les opérations des séquences *communes* et *mutables*. Les listes fournissent également la méthode supplémentaire suivante :

sort(*, *key=None*, *reverse=False*)

Cette méthode trie la liste sur place, en utilisant uniquement des comparaisons < entre les éléments. Les exceptions ne sont pas supprimées : si n'importe quelle opération de comparaison échoue, le tri échoue (et la liste sera probablement laissée dans un état partiellement modifié).

`sort()` accepte deux arguments qui ne peuvent être fournis que nommés (voir *arguments nommés*) :

key spécifie une fonction d'un argument utilisée pour extraire une clé de comparaison de chaque élément de la liste (par exemple, `key=str.lower`). La clé correspondant à chaque élément de la liste n'est calculée qu'une seule fois, puis utilisée durant tout le processus. La valeur par défaut, `None`, signifie que les éléments sont triés directement sans calculer de « valeur clé » séparée.

La fonction utilitaire `functools.cmp_to_key()` est disponible pour convertir une fonction *cmp* du style 2.x à une fonction *key*.

reverse, une valeur booléenne. Si elle est `True`, la liste d'éléments est triée comme si toutes les comparaisons étaient inversées.

Cette méthode modifie la séquence sur place pour économiser de l'espace lors du tri de grandes séquences. Pour rappeler aux utilisateurs cet effet de bord, elle ne renvoie pas la séquence triée (utilisez `sorted()` pour demander explicitement une nouvelle instance de liste triée).

La méthode `sort()` est garantie stable. Un tri est stable s'il garantit de ne pas changer l'ordre relatif des éléments égaux — cela est utile pour trier en plusieurs passes (par exemple, trier par service, puis par niveau de salaire).

Pour des exemples de tris et un bref tutoriel, consultez sortinghowto.

l'effet de tenter de modifier, ou même inspecter la liste pendant qu'on la trie est indéfini. L'implémentation C de Python fait apparaître la liste comme vide pour la durée du traitement, et lève `ValueError` si elle détecte que la liste a été modifiée au cours du tri.

4.7.5 *N*-uplets

Les *n*-uplets (*tuples* en anglais) sont des séquences immuables, généralement utilisées pour stocker des collections de données hétérogènes (telles que les paires produites par la fonction native `enumerate()`). Les *n*-uplets sont également utilisés dans des cas où une séquence homogène et immuable de données est nécessaire (pour, par exemple, les stocker dans un `ensemble` ou un `dictionnaire`).

class tuple([*iterable*])

Les *n*-uplets peuvent être construits de différentes façons :

— en utilisant une paire de parenthèses pour désigner le *n*-uplet vide : `()` ;
— en utilisant une virgule, pour créer un *n*-uplet d'un élément : `a,` ou `(a,)` ;
— en séparant les éléments avec des virgules : `a, b, c` ou `(a, b, c)` ;
— en utilisant la fonction native `tuple()` : `tuple()` ou `tuple(iterable)`.

Le constructeur construit un *n*-uplet dont les éléments sont les mêmes et dans le même ordre que les éléments de *iterable*. *iterable* peut être une séquence, un conteneur qui prend en charge l'itération ou un itérateur. Si *iterable* est déjà un *n*-uplet, il est renvoyé inchangé. Par exemple, `tuple('abc')` renvoie `('a', 'b', 'c')` et `tuple( [1, 2, 3] )` renvoie `(1, 2, 3)`. Si aucun argument n'est donné, le constructeur crée un nouveau *n*-uplet vide, `()`.

Notez que c'est en fait la virgule qui fait un *n*-uplet et non les parenthèses. Les parenthèses sont facultatives, sauf dans le cas du *n*-uplet vide, ou lorsqu'elles sont nécessaires pour éviter l'ambiguïté syntaxique. Par exemple,

`f(a, b, c)` est un appel de fonction avec trois arguments, alors que `f((a, b, c))` est un appel de fonction avec un triplet comme unique argument.

Les *n*-uplets implémentent toutes les opérations *communes* des séquences.

Pour les collections hétérogènes de données où l'accès par nom est plus clair que l'accès par indice, `collections.namedtuple()` peut être un choix plus approprié qu'un simple *n*-uplet.

4.7.6 Ranges

Le type `range` représente une séquence immuable de nombres et est couramment utilisé pour itérer un certain nombre de fois dans les boucles `for`.

class range(*stop*)

class range(*start*, *stop*[, *step*])

Les arguments du constructeur de *range* doivent être des entiers (des `int` ou tout autre objet qui implémente la méthode spéciale `__index__()`). La valeur par défaut de l'argument *step* est `1`. La valeur par défaut de l'argument *start* est `0`. Si *step* est égal à zéro, une exception `ValueError` est levée.

Pour un *step* positif, le contenu d'un *range* `r` est déterminé par la formule `r[i] = start + step*i` où `i >= 0` et `r[i] < stop`.

Pour un *step* négatif, le contenu du *range* est toujours déterminé par la formule `r[i] = start + step*i`, mais les contraintes sont `i >= 0` et `r[i] > stop`.

Un objet *range* sera vide si `r[0]` ne répond pas à la contrainte de valeur. Les *range* prennent en charge les indices négatifs, mais ceux-ci sont interprétées comme un indice à partir de la fin de la séquence déterminée par les indices positifs.

Les *range* contenant des valeurs absolues plus grandes que `sys.maxsize` sont permises, mais certaines fonctionnalités (comme `len()`) peuvent lever `OverflowError`.

Exemples avec *range* :

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(1, 11))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> list(range(0, 30, 5))
[0, 5, 10, 15, 20, 25]
>>> list(range(0, 10, 3))
[0, 3, 6, 9]
>>> list(range(0, -10, -1))
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
>>> list(range(0))
[]
>>> list(range(1, 0))
[]
```

range implémente toutes les opérations *communes* des séquences sauf la concaténation et la répétition (en raison du fait que les *range* ne peuvent représenter que des séquences qui respectent un motif strict et que la répétition et la concaténation les feraient dévier de ce motif).

start

Valeur du paramètre *start* (ou `0` si le paramètre n'a pas été fourni)

stop

Valeur du paramètre *stop*

step

Valeur du paramètre *step* (ou `1` si le paramètre n'a pas été fourni)

L'avantage du type `range` sur une `liste` classique ou un `n-uplet` est qu'un objet `range` occupe toujours la même

(petite) quantité de mémoire, peu importe la taille de l'intervalle qu'il représente (car il ne stocke que les valeurs `start`, `stop`, `step`, le calcul des éléments individuels et les sous-intervalles au besoin).

Les *range* implémentent la classe mère abstraite `collections.abc.Sequence` et offrent des fonctionnalités telles que les tests d'appartenance (avec *in*), de recherche par indice, les tranches et ils gèrent les indices négatifs (voir *Types séquentiels — list, tuple, range*) :

```
>>> r = range(0, 20, 2)
>>> r
range(0, 20, 2)
>>> 11 in r
False
>>> 10 in r
True
>>> r.index(10)
5
>>> r[5]
10
>>> r[:5]
range(0, 10, 2)
>>> r[-1]
18
```

Comparer des *range* avec `==` et `!=` les compare comme des séquences. C'est-à-dire que deux objets *range* sont considérés comme égaux s'ils représentent la même séquence de valeurs. (Notez que deux objets *range* dits égaux pourraient avoir leurs attributs `start`, `stop` et `step` différents, par exemple `range(0) == range(2, 1, 3)` ou `range(0, 3, 2) == range(0, 4, 2)`.)

Modifié dans la version 3.2 : implémente la classe mère abstraite *Sequence*. prend en charge les tranches (*slicing*) et les indices négatifs. Teste l'appartenance d'un `int` en temps constant au lieu d'itérer sur tous les éléments.

Modifié dans la version 3.3 : `==` et `!=` comparent des *range* en fonction de la séquence de valeurs qu'ils définissent (au lieu d'une comparaison fondée sur l'identité de l'objet).

Added the `start`, `stop` and `step` attributes.

Voir aussi

— The linspace recipe shows how to implement a lazy version of range suitable for floating-point applications.

4.8 Text and Binary Sequence Type Methods Summary

The following table summarizes the text and binary sequence types methods by category.

Category		`str` methods	
Formatting		`str.format()`	
		`str.format_map()`	
		f-strings	
		Formatage de chaines à la printf	
Searching and Replacing	`str.find()`		`str.rfind()`
	`str.index()`		`str.rindex()`
		`str.startswith()`	
		`str.endswith()`	
		`str.count()`	
		`str.replace()`	
Splitting and Joining	`str.split()`		`str.rsplit()`

Category		str methods	
		str.splitlines()	
		str.partition()	
		str.rpartition()	
		str.join()	
String Classification		str.isalpha()	
		str.isdecimal()	
		str.isdigit()	
		str.isnumeric()	
		str.isalnum()	
		str.isidentifier()	
		str.islower()	
		str.isupper()	
		str.istitle()	
		str.isspace()	
		str.isprintable()	
Case Manipulation		str.lower()	
		str.upper()	
		str.casefold()	
		str.capitalize()	
		str.title()	
		str.swapcase()	
Padding and Stripping	str.ljust()		str.rjust()
		str.center()	
		str.expandtabs()	
		str.strip()	
	str.lstrip()		
Translation and Encoding		str.translate()	
		str.maketrans()	
		str.encode()	

4.9 Type Séquence de Texte — str

Les données textuelles en Python sont manipulées avec des objets `str` ou *strings*. Les chaînes sont des *séquences* immuables de points de code Unicode. Les chaînes littérales peuvent être écrites de différentes manières :

- entre guillemets simples : `'cela autorise les "guillemets anglais"'`;
- entre guillemets (anglais) : `"cela autorise les guillemets 'simples'"`;
- entre guillemets triples : `'''Trois guillemets simples'''`, `"""Trois guillemets anglais"""`.

Les chaînes entre guillemets triples peuvent couvrir plusieurs lignes, tous les espaces associées sont alors incluses dans la chaîne littérale.

Les chaînes littérales qui font partie d'une seule expression et ont seulement des espaces entre elles sont implicitement converties en une seule chaîne littérale. Autrement dit, `("spam " "eggs") == "spam eggs"`.

See strings for more about the various forms of string literal, including supported escape sequences, and the `r` ("raw") prefix that disables most escape sequence processing.

Les chaînes peuvent également être créées à partir d'autres objets à l'aide du constructeur `str`.

Comme il n'y a pas de type « caractère » propre, un indice d'une chaîne produit une chaîne de longueur 1. Autrement dit, pour une chaîne non vide *s*, `s[0] == s[0:1]`.

Il n'y a aucun type de chaîne mutable, mais `str.join()` ou `io.StringIO` peuvent être utilisées pour construire efficacement des chaînes à partir de plusieurs fragments.

Modifié dans la version 3.3 : pour une compatibilité ascendante avec la série Python 2, le préfixe `u` est à nouveau

autorisé sur les chaînes littérales. Il n'a aucun effet sur le sens des chaînes littérales et ne peut pas être combiné avec le préfixe `r`.

class str(*object=''*)

class str(*object=b''*, *encoding='utf-8'*, *errors='strict'*)

Renvoie une représentation en *chaîne de caractères* de *object*. Si *object* n'est pas fourni, renvoie une chaîne vide. Sinon, le comportement de `str()` dépend des valeurs données pour *encoding* et *errors*, comme indiqué ci-dessous.

If neither *encoding* nor *errors* is given, `str(object)` returns `type(object).__str__(object)`, which is the "informal" or nicely printable string representation of *object*. For string objects, this is the string itself. If *object* does not have a `__str__()` method, then `str()` falls back to returning `repr(object)`.

Si au moins un des deux arguments *encoding* ou *errors* est donné, *object* doit être un *objet octet-compatible* (par exemple, `bytes` ou `bytearray`). Dans ce cas, si *object* est un objet `bytes` (ou `bytearray`), alors `str(bytes, encoding, errors)` est équivalent à `bytes.decode(encoding, errors)`. Sinon, l'objet *bytes* sous-jacent au tampon est obtenu avant d'appeler `bytes.decode()`. Voir *Séquences Binaires — bytes, bytearray, vue mémoire* et bufferobjects pour plus d'informations sur les tampons.

Donner un objet `bytes` à `str()` sans argument *encoding* ni argument *errors* relève du premier cas, où la représentation informelle de la chaîne est renvoyée (voir aussi l'option `-b` de Python). Par exemple :

```
>>> str(b'Zoot!')
"b'Zoot!'"
```

Pour plus d'informations sur la classe `str` et ses méthodes, voir les sections *Type Séquence de Texte — str* et *Méthodes de chaînes de caractères*. Pour formater des chaînes de caractères, voir les sections f-strings et *Syntaxe de formatage de chaîne*. La section *Services de Manipulation de Texte* contient aussi des informations.

4.9.1 Méthodes de chaînes de caractères

Les chaînes implémentent toutes les opérations *communes des séquences*, ainsi que les autres méthodes décrites ci-dessous.

Les chaînes gèrent aussi deux styles de mise en forme, l'un fournissant une grande flexibilité et de personnalisation (voir `str.format()`, *Syntaxe de formatage de chaîne* et *Formatage personnalisé de chaîne*) et l'autre basée sur le style de formatage de `printf` du C qui gère une gamme plus étroite de types et est légèrement plus difficile à utiliser correctement, mais qui est souvent plus rapide pour les cas pris en charge (*Formatage de chaines à la printf*).

La section *Services de Manipulation de Texte* de la bibliothèque standard couvre un certain nombre d'autres modules qui fournissent différents services relatifs au texte (y compris les expressions rationnelles dans le module `re`).

`str`.**capitalize**()

Renvoie une copie de la chaîne avec son premier caractère en majuscule et le reste en minuscule.

Modifié dans la version 3.8 : le premier caractère est maintenant mis en *titlecase* plutôt qu'en majuscule. Cela veut dire que les caractères comme les digrammes auront seulement leur première lettre en majuscule, au lieu du caractère en entier.

`str`.**casefold**()

Renvoie une copie *casefolded* de la chaîne. Les chaînes *casefolded* peuvent être utilisées dans des comparaisons insensibles à la casse.

Le *casefolding* est une technique agressive de mise en minuscule, car il vise à éliminer toutes les distinctions de casse dans une chaîne. Par exemple, la lettre minuscule `'ß'` de l'allemand équivaut à `"ss"`. Comme il est déjà minuscule, `lower()` ne fait rien à `'ß'` ; `casefold()` le convertit en `"ss"`.

The casefolding algorithm is described in section 3.13 'Default Case Folding' of the Unicode Standard.

Ajouté dans la version 3.3.

`str`.**center**(*width*[, *fillchar*])

Renvoie la chaîne au centre d'une chaîne de longueur *width*. Le remplissage est fait en utilisant l'argument

fillchar (qui par défaut est une espace ASCII). La chaîne d'origine est renvoyée si *width* est inférieure ou égale à `len(s)`.

`str.count`(*sub*[, *start*[, *end*]])

Renvoie le nombre d'occurrences de *sub* ne se chevauchant pas dans l'intervalle [*start*, *end*]. Les arguments facultatifs *start* et *end* sont interprétés comme dans la notation des tranches (*slices* en anglais).

Si *sub* est vide, renvoie le nombre de chaînes vides entre les caractères de début et de fin, ce qui correspond à la longueur de la chaîne plus un.

`str.encode`(*encoding='utf-8'*, *errors='strict'*)

Renvoie la chaine encodée dans une instance de `bytes`.

encoding vaut par défaut `utf-8` ; pour une liste des encodages possibles, voir la section *Standard Encodings*.

errors détermine la manière dont les erreurs sont traitées. La valeur par défaut est `'strict'`, ce qui signifie que les erreurs d'encodage lèvent une `UnicodeError`. Les autres valeurs possibles sont `'ignore'`, `'replace'`, `'xmlcharrefreplace'`, `'backslashreplace'` et tout autre nom enregistré *via* `codecs.register_error()`. Voir la section *Gestionnaires d'erreurs* pour plus de détails.

Pour des raisons de performances, la valeur de *errors* n'est pas vérifiée à moins qu'une erreur d'encodage ne se produise réellement, que le *mode développeur* ne soit activé ou que Python ait été compilé en mode débogage.

Modifié dans la version 3.1 : gère les arguments nommés.

Modifié dans la version 3.9 : les valeurs de *errors* sont maintenant vérifiées en *mode de développement* et en mode de débogage.

`str.endswith`(*suffix*[, *start*[, *end*]])

Renvoie `True` si la chaîne se termine par *suffix*, sinon `False`. *suffix* peut aussi être un *n*-uplet de suffixes à rechercher. Si l'argument optionnel *start* est donné, le test se fait à partir de cette position. Si l'argument optionnel *end* est fourni, la comparaison s'arrête à cette position.

`str.expandtabs`(*tabsize=8*)

Renvoie une copie de la chaîne où toutes les tabulations sont remplacées par une ou plusieurs espaces, en fonction de la colonne courante et de la taille de tabulation donnée. Les positions des tabulations se trouvent tous les *tabsize* caractères (8 par défaut, ce qui donne les positions de tabulations aux colonnes 0, 8, 16 et ainsi de suite). Pour travailler sur la chaîne, la colonne en cours est mise à zéro et la chaîne est examinée caractère par caractère. Si le caractère est une tabulation (`\t`), un ou plusieurs caractères d'espacement sont insérés dans le résultat jusqu'à ce que la colonne courante soit égale à la position de tabulation suivante (le caractère tabulation lui-même n'est pas copié). Si le caractère est un saut de ligne (`\n`) ou un retour chariot (`\r`), il est copié et la colonne en cours est remise à zéro. Tout autre caractère est copié inchangé et la colonne en cours est incrémentée de un indépendamment de la façon dont le caractère est représenté lors de l'affichage.

```
>>> '01\t012\t0123\t01234'.expandtabs()
'01      012     0123    01234'
>>> '01\t012\t0123\t01234'.expandtabs(4)
'01  012 0123    01234'
```

`str.find`(*sub*[, *start*[, *end*]])

Renvoie l'indice de la première position dans la chaîne où *sub* est trouvé dans le découpage `s[start:end]`. Les arguments facultatifs *start* et *end* sont interprétés comme dans la notation des découpages (*slice* en anglais). Renvoie `-1` si *sub* n'est pas trouvé.

Note

la méthode `find()` ne doit être utilisée que si vous avez besoin de connaître la position de *sub*. Pour vérifier si *sub* est une sous chaine ou non, utilisez l'opérateur `in` :

```
>>> 'Py' in 'Python'
True
```

`str.format(*args, **kwargs)`

Formate une chaîne. La chaîne sur laquelle cette méthode est appelée peut contenir du texte littéral ou des emplacements de remplacement délimités par des accolades `{}`. Chaque champ de remplacement contient soit l'indice numérique d'un argument positionnel, soit le nom d'un argument nommé. Renvoie une copie de la chaîne où chaque champ de remplacement est remplacé par la valeur de chaîne de l'argument correspondant.

```
>>> "The sum of 1 + 2 is {0}".format(1+2)
'The sum of 1 + 2 is 3'
```

Voir *Syntaxe de formatage de chaîne* pour une description des options de formatage qui peuvent être spécifiées dans les chaînes de format.

Note

lors du formatage avec le format `n` (comme `'{:n}'.format(1234)`) d'un nombre (`int`, `float`, `complex`, `decimal.Decimal` et classes dérivées), la fonction met temporairement la variable `LC_CTYPE` à la valeur de `LC_NUMERIC` pour décoder correctement les attributs `decimal_point` et `thousands_sep` de `localeconv()`, s'ils ne sont pas en ASCII ou font plus d'un octet et que `LC_NUMERIC` est différent de `LC_CTYPE`. Ce changement temporaire affecte les autres fils d'exécution.

Modifié dans la version 3.7 : lors du formatage d'un nombre avec le format `n`, la fonction change temporairement `LC_CTYPE` par la valeur de `LC_NUMERIC` dans certains cas.

`str.format_map(mapping, /)`

Semblable à `str.format(**mapping)`, sauf que `mapping` est utilisé directement et non copié dans un `dict`. C'est utile si, par exemple, `mapping` est une sous-classe de `dict` :

```
>>> class Default(dict):
...     def __missing__(self, key):
...         return key
...
>>> '{name} was born in {country}'.format_map(Default(name='Guido'))
'Guido was born in country'
```

Ajouté dans la version 3.2.

`str.index(sub[, start[, end]])`

Comme `find()`, mais lève une `ValueError` lorsque la chaîne est introuvable.

`str.isalnum()`

Renvoie `True` si tous les caractères de la chaîne sont alphanumériques et qu'il y a au moins un caractère, sinon `False`. Un caractère `c` est alphanumérique si l'un des tests suivants renvoie `True` : `c.isalpha()`, `c.isdecimal()`, `c.isdigit()` ou `c.isnumeric()`.

`str.isalpha()`

Return `True` if all characters in the string are alphabetic and there is at least one character, `False` otherwise. Alphabetic characters are those characters defined in the Unicode character database as "Letter", i.e., those with general category property being one of "Lm", "Lt", "Lu", "Ll", or "Lo". Note that this is different from the Alphabetic property defined in the section 4.10 'Letters, Alphabetic, and Ideographic' of the Unicode Standard.

`str.isascii()`

Renvoie `True` si la chaîne est vide ou ne contient que des caractères ASCII, `False` sinon. Les caractères ASCII ont un code dans l'intervalle `"U+0000"-"U+007F"`.

Ajouté dans la version 3.7.

`str.isdecimal()`

Renvoie `True` si tous les caractères de la chaîne sont des caractères décimaux et qu'elle contient au moins un caractère, sinon `False`. Les caractères décimaux sont ceux pouvant être utilisés pour former des nombres

en base 10, tels que U+0660, ARABIC-INDIC DIGIT ZERO. Formellement, un caractère décimal est un caractère dans la catégorie Unicode générale "Nd".

`str.isdigit()`

Renvoie `True` si tous les caractères de la chaîne sont des chiffres et qu'elle contient au moins un caractère, `False` sinon. Les chiffres incluent des caractères décimaux et des chiffres qui nécessitent une manipulation particulière, tels que les *compatibility superscript digits*. Ça couvre les chiffres qui ne peuvent pas être utilisés pour construire des nombres en base 10, tels que les nombres de Kharosthi. Formellement, un chiffre est un caractère dont la valeur de la propriété *Numeric_Type* est *Digit* ou *Decimal*.

`str.isidentifier()`

Renvoie `True` si la chaîne est un identifiant valide selon la définition du langage, section identifiers.

`keyword.iskeyword()` can be used to test whether string `s` is a reserved identifier, such as `def` and `class`.

Par exemple :

```
>>> from keyword import iskeyword

>>> 'hello'.isidentifier(), iskeyword('hello')
(True, False)
>>> 'def'.isidentifier(), iskeyword('def')
(True, True)
```

`str.islower()`

Renvoie `True` si tous les caractères capitalisables[4] de la chaîne sont en minuscules et qu'elle contient au moins un caractère capitalisable. Renvoie `False` dans le cas contraire.

`str.isnumeric()`

Renvoie `True` si tous les caractères de la chaîne sont des caractères numériques et qu'elle contient au moins un caractère, sinon `False`. Les caractères numériques comprennent les chiffres et tous les caractères qui ont la propriété Unicode *numeric value*, par exemple U+2155, *VULGAR FRACTION OF FIFTH*. Formellement, les caractères numériques sont ceux avec les propriétés *Numeric_Type=Digit*, *Numeric_Type=Decimal* ou *Numeric_Type=Numeric*.

`str.isprintable()`

Return true if all characters in the string are printable, false if it contains at least one non-printable character.

Here "printable" means the character is suitable for `repr()` to use in its output ; "non-printable" means that `repr()` on built-in types will hex-escape the character. It has no bearing on the handling of strings written to `sys.stdout` or `sys.stderr`.

The printable characters are those which in the Unicode character database (see `unicodedata`) have a general category in group Letter, Mark, Number, Punctuation, or Symbol (L, M, N, P, or S) ; plus the ASCII space 0x20. Nonprintable characters are those in group Separator or Other (Z or C), except the ASCII space.

`str.isspace()`

Renvoie `True` s'il n'y a que des caractères d'espacement dans la chaîne et qu'elle comporte au moins un caractère. Renvoie `False` dans le cas contraire.

Un caractère est considéré comme un caractère d'espacement (*whitespace* en anglais) si, dans la base de données caractères Unicode (voir `unicodedata`), sa catégorie générale est `Zs` (« séparateur, espace »), ou sa classe bidirectionnelle est une de `WS`, `B`, ou `S`.

`str.istitle()`

Renvoie `True` si la chaîne est une chaîne *titlecased* et qu'elle contient au moins un caractère, par exemple les caractères majuscules ne peuvent suivre les caractères non capitalisables et les caractères minuscules ne peuvent suivre que des caractères capitalisables. Renvoie `False` dans le cas contraire.

4. Les caractères capitalisables sont ceux dont la propriété Unicode *general category* est soit "Lu" (pour *Letter*, *uppercase*), soit "Ll" (pour *Letter*, *lowercase*), soit "Lt" (pour *Letter*, *titlecase*).

`str.isupper()`

Renvoie `True` si tous les caractères différentiables sur la casse[page 57, 4] de la chaîne sont en majuscules et s'il y a au moins un caractère différentiable sur la casse, sinon `False`.

```
>>> 'BANANA'.isupper()
True
>>> 'banana'.isupper()
False
>>> 'baNana'.isupper()
False
>>> ' '.isupper()
False
```

`str.join(iterable)`

Renvoie une chaîne qui est la concaténation des chaînes contenues dans *iterable*. Une `TypeError` est levée si une valeur d'*iterable* n'est pas une chaîne, y compris pour les objets `bytes`. Le séparateur entre les éléments est la chaîne fournissant cette méthode.

`str.ljust(width[, fillchar])`

Renvoie la chaîne justifiée à gauche dans une chaîne de longueur *width*. Le bourrage est fait en utilisant *fillchar* (qui par défaut est une espace ASCII). La chaîne d'origine est renvoyée si *width* est inférieure ou égale à `len(s)`.

`str.lower()`

Renvoie une copie de la chaîne avec tous les caractères différentiables sur la casse[page 57, 4] convertis en minuscules.

The lowercasing algorithm used is described in section 3.13 'Default Case Folding' of the Unicode Standard.

`str.lstrip([chars])`

Renvoie une copie de la chaîne avec des caractères supprimés au début. L'argument *chars* est une chaîne spécifiant le jeu de caractères à supprimer. En cas d'omission ou `None`, la valeur par défaut de *chars* permet de supprimer des caractères d'espacement. L'argument *chars* n'est pas un préfixe, toutes les combinaisons de ses valeurs sont supprimées :

```
>>> '   spacious   '.lstrip()
'spacious   '
>>> 'www.example.com'.lstrip('cmowz.')
'example.com'
```

Voir `str.removeprefix()` pour une méthode qui supprime une seule chaîne de préfixe plutôt que la totalité d'un ensemble de caractères. Par exemple :

```
>>> 'Arthur: three!'.lstrip('Arthur: ')
'ee!'
>>> 'Arthur: three!'.removeprefix('Arthur: ')
'three!'
```

static `str.maketrans(x[, y[, z]])`

Cette méthode statique renvoie une table de traduction utilisable pour `str.translate()`.

Si un seul argument est fourni, ce soit être un dictionnaire faisant correspondre des points de code Unicode (nombres entiers) ou des caractères (chaînes de longueur 1) à des points de code Unicode.

Si deux arguments sont fournis, ce doit être deux chaînes de caractères de même longueur. Le dictionnaire renvoyé fera correspondre pour chaque caractère de x un caractère de y pris à la même place. Si un troisième argument est fourni, ce doit être une chaîne dont chaque caractère correspondra à `None` dans le résultat.

`str.partition(sep)`

Divise la chaîne à la première occurrence de *sep*, et renvoie un triplet contenant la partie avant le séparateur,

le séparateur lui-même, et la partie après le séparateur. Si le séparateur n'est pas trouvé, le triplet contient la chaîne elle-même, suivie de deux chaînes vides.

`str.removeprefix(prefix, /)`

Si la chaîne de caractères commence par la chaîne *prefix*, renvoie `string[len(prefix):]`. Sinon, renvoie une copie de la chaîne originale :

```
>>> 'TestHook'.removeprefix('Test')
'Hook'
>>> 'BaseTestCase'.removeprefix('Test')
'BaseTestCase'
```

Ajouté dans la version 3.9.

`str.removesuffix(suffix, /)`

Si la chaîne de caractères se termine par la chaîne *suffix* et que *suffix* n'est pas vide, renvoie `string[:-len(suffix)]`. Sinon, renvoie une copie de la chaîne originale :

```
>>> 'MiscTests'.removesuffix('Tests')
'Misc'
>>> 'TmpDirMixin'.removesuffix('Tests')
'TmpDirMixin'
```

Ajouté dans la version 3.9.

`str.replace(old, new, count=-1)`

Return a copy of the string with all occurrences of substring *old* replaced by *new*. If *count* is given, only the first *count* occurrences are replaced. If *count* is not specified or `-1`, then all occurrences are replaced.

Modifié dans la version 3.13 : *count* is now supported as a keyword argument.

`str.rfind(sub[, start[, end]])`

Renvoie l'indice le plus élevé dans la chaîne où la sous-chaîne *sub* se trouve, de telle sorte que *sub* soit contenue dans `s[start:end]`. Les arguments facultatifs *start* et *end* sont interprétés comme dans la notation des découpages. Renvoie `-1` en cas d'échec.

`str.rindex(sub[, start[, end]])`

Comme `rfind()` mais lève une exception `ValueError` lorsque la sous-chaîne *sub* est introuvable.

`str.rjust(width[, fillchar])`

Renvoie la chaîne justifiée à droite dans une chaîne de longueur *width*. Le bourrage est fait en utilisant le caractère spécifié par *fillchar* (par défaut une espace ASCII). La chaîne d'origine est renvoyée si *width* est inférieure ou égale à `len(s)`.

`str.rpartition(sep)`

Divise la chaîne à la dernière occurrence de *sep*, et renvoie un triplet contenant la partie avant le séparateur, le séparateur lui-même, et la partie après le séparateur. Si le séparateur n'est pas trouvé, le triplet contient deux chaînes vides, puis la chaîne elle-même.

`str.rsplit(sep=None, maxsplit=-1)`

Renvoie une liste des mots de la chaîne, en utilisant *sep* comme séparateur. Si *maxsplit* est donné, c'est le nombre maximum de divisions qui pourront être faites, celles « les plus à droite ». Si *sep* est pas spécifié ou est `None`, tout caractère d'espacement est un séparateur. En dehors du fait qu'il découpe par la droite, `rsplit()` se comporte comme `split()` qui est décrit en détail ci-dessous.

`str.rstrip([chars])`

Renvoie une copie de la chaîne avec des caractères finaux supprimés. L'argument *chars* est une chaîne spécifiant le jeu de caractères à supprimer. En cas d'omission ou `None`, les caractères d'espacement sont supprimés. L'argument *chars* n'est pas un suffixe : toutes les combinaisons de ses valeurs sont retirées :

```
>>> '   spacious   '.rstrip()
'   spacious'
>>> 'mississippi'.rstrip('ipz')
'mississ'
```

Voir `str.removesuffix()` pour une méthode qui supprime une seule chaîne de suffixe plutôt que la totalité d'un ensemble de caractères. Par exemple :

```
>>> 'Monty Python'.rstrip(' Python')
'M'
>>> 'Monty Python'.removesuffix(' Python')
'Monty'
```

`str.`**`split`**(*sep=None*, *maxsplit=-1*)

Renvoie une liste des mots de la chaîne, en utilisant *sep* comme séparateur de mots. Si *maxsplit* est donné, c'est le nombre maximum de divisions qui pourront être effectuées (donnant ainsi une liste de longueur `maxsplit+1`). Si *maxsplit* n'est pas fourni, ou vaut `-1`, le nombre de découpes n'est pas limité (toutes les découpes possibles sont faites).

If *sep* is given, consecutive delimiters are not grouped together and are deemed to delimit empty strings (for example, `'1,,2'.split(',')` returns `['1', '', '2']`). The *sep* argument may consist of multiple characters as a single delimiter (to split with multiple delimiters, use `re.split()`). Splitting an empty string with a specified separator returns `['']`.

Par exemple :

```
>>> '1,2,3'.split(',')
['1', '2', '3']
>>> '1,2,3'.split(',', maxsplit=1)
['1', '2,3']
>>> '1,2,,3,'.split(',')
['1', '2', '', '3', '']
>>> '1<>2<>3<4'.split('<>')
['1', '2', '3<4']
```

Si *sep* n'est pas spécifié ou est `None`, un autre algorithme de découpage est appliqué : les caractères d'espacement consécutifs sont considérés comme un seul séparateur, et le résultat ne contient pas les chaînes vides de début ou de la fin si la chaîne est préfixée ou suffixé de caractères d'espacement. Par conséquent, diviser une chaîne vide ou une chaîne composée d'espaces avec un séparateur `None` renvoie `[]`.

Par exemple :

```
>>> '1 2 3'.split()
['1', '2', '3']
>>> '1 2 3'.split(maxsplit=1)
['1', '2 3']
>>> '   1   2   3   '.split()
['1', '2', '3']
```

`str.`**`splitlines`**(*keepends=False*)

Renvoie les lignes de la chaîne sous forme de liste, la découpe se faisant au niveau des limites des lignes. Les sauts de ligne ne sont pas inclus dans la liste des résultats, sauf si *keepends* est donné et est vrai.

Cette méthode découpe sur les limites de ligne suivantes. Ces limites sont un sur-ensemble de *universal newlines*.

Représentation	Description
`\n`	Saut de ligne
`\r`	Retour chariot
`\r\n`	Retour chariot + saut de ligne
`\v` or `\x0b`	Tabulation verticale
`\f` or `\x0c`	Saut de page
`\x1c`	Séparateur de fichiers
`\x1d`	Séparateur de groupes
`\x1e`	Séparateur d'enregistrements
`\x85`	Ligne suivante (code de contrôle *C1*)
`\u2028`	Séparateur de ligne
`\u2029`	Séparateur de paragraphe

Modifié dans la version 3.2 : `\v` et `\f` ajoutés à la liste des limites de lignes.

Par exemple :

```
>>> 'ab c\n\nde fg\rkl\r\n'.splitlines()
['ab c', '', 'de fg', 'kl']
>>> 'ab c\n\nde fg\rkl\r\n'.splitlines(keepends=True)
['ab c\n', '\n', 'de fg\r', 'kl\r\n']
```

Contrairement à *split()* lorsque *sep* est fourni, cette méthode renvoie une liste vide pour la chaîne vide, et un saut de ligne à la fin ne se traduit pas par une ligne supplémentaire :

```
>>> "".splitlines()
[]
>>> "One line\n".splitlines()
['One line']
```

À titre de comparaison, `split('\n')` donne :

```
>>> ''.split('\n')
['']
>>> 'Two lines\n'.split('\n')
['Two lines', '']
```

`str.`**`startswith`**(*prefix*[, *start*[, *end*]])

Renvoie `True` si la chaîne commence par *prefix*, sinon `False`. *prefix* peut aussi être un *n*-uplet de préfixes à rechercher. Lorsque *start* est donné, la comparaison commence à cette position et, lorsque *end* est donné, la comparaison s'arrête à celle-ci.

`str.`**`strip`**([*chars*])

Renvoie une copie de la chaîne dont des caractères initiaux et finaux sont supprimés. L'argument *chars* est une chaîne spécifiant le jeu de caractères à supprimer. En cas d'omission ou `None`, les caractères d'espacement sont supprimés. L'argument *chars* est pas un préfixe ni un suffixe, toutes les combinaisons de ses valeurs sont supprimées :

```
>>> '   spacious   '.strip()
'spacious'
>>> 'www.example.com'.strip('cmowz.')
'example'
```

Les caractères de *char* sont retirés du début et de la fin de la chaîne. Les caractères sont retirés de la gauche jusqu'à atteindre un caractère ne figurant pas dans le jeu de caractères dans *chars*. La même opération a lieu par la droite. Par exemple :

```
>>> comment_string = '#....... Section 3.2.1 Issue #32 .......'
>>> comment_string.strip('.#! ')
'Section 3.2.1 Issue #32'
```

str.**swapcase**()

Renvoie une copie de la chaîne dont les caractères majuscules sont convertis en minuscules et vice versa. Notez qu'il est pas nécessairement vrai que `s.swapcase().swapcase() == s`.

str.**title**()

Renvoie une version de la chaîne où les mots commencent par une capitale et les caractères restants sont en minuscules.

Par exemple :

```
>>> 'Hello world'.title()
'Hello World'
```

Pour l'algorithme, la notion de mot est définie simplement et indépendamment de la langue comme un groupe de lettres consécutives. La définition fonctionne dans de nombreux contextes, mais cela signifie que les apostrophes (typiquement de la forme possessive en Anglais) forment les limites de mot, ce qui n'est pas toujours le résultat souhaité :

```
>>> "they're bill's friends from the UK".title()
"They're Bill'S Friends From The Uk"
```

La fonction *`string.capwords()`* n'a pas ce problème, car elle sépare les mots uniquement sur les espaces.

Sinon, une solution pour contourner le problème des apostrophes est d'utiliser des expressions rationnelles :

```
>>> import re
>>> def titlecase(s):
...     return re.sub(r"[A-Za-z]+('[A-Za-z]+)?",
...                   lambda mo: mo.group(0).capitalize(),
...                   s)
...
>>> titlecase("they're bill's friends.")
"They're Bill's Friends."
```

str.**translate**(*table*)

Return a copy of the string in which each character has been mapped through the given translation table. The table must be an object that implements indexing via `__getitem__()`, typically a *mapping* or *sequence*. When indexed by a Unicode ordinal (an integer), the table object can do any of the following : return a Unicode ordinal or a string, to map the character to one or more other characters ; return `None`, to delete the character from the return string ; or raise a *`LookupError`* exception, to map the character to itself.

Vous pouvez utiliser *`str.maketrans()`* pour créer une table de correspondances de caractères dans différents formats.

Voir aussi le module *`codecs`* pour une approche plus souple de changements de caractères par correspondance.

str.**upper**()

Renvoie une copie de la chaîne où tous les caractères capitalisables[page 57, 4] ont été convertis en capitales. Notez que `s.upper().isupper()` peut être `False` si `s` contient des caractères non capitalisables ou si la catégorie Unicode d'un caractère du résultat n'est pas "Lu" (*Letter*, *uppercase*), mais par exemple "Lt" (*Letter*, *titlecase*).

The uppercasing algorithm used is described in section 3.13 'Default Case Folding' of the Unicode Standard.

str.**zfill**(*width*)

Renvoie une copie de la chaîne remplie par la gauche du chiffre (le caractère ASCII) `'0'` pour faire une chaîne de longueur *width*. Un préfixe (`'+'` / `'-'`) est permis par l'insertion du caractère de bourrage *après* le caractère désigné plutôt qu'avant. La chaîne d'origine est renvoyée si *width* est inférieure ou égale à `len(s)`.

Par exemple :

```
>>> "42".zfill(5)
'00042'
>>> "-42".zfill(5)
'-0042'
```

4.9.2 Formatage de chaines à la `printf`

> **Note**
>
> ces opérations de mise en forme contiennent des bizarreries menant à de nombreuses erreurs classiques (telles que ne pas réussir à afficher des *n*-uplets ou des dictionnaires correctement). Utiliser les formatted string literals, la méthode `str.format()` ou les *template strings* aide à éviter ces erreurs. Chacune de ces alternatives apporte son lot d'avantages et inconvénients en matière de simplicité, de flexibilité et/ou de généralisation possible.

String objects have one unique built-in operation : the `%` operator (modulo). This is also known as the string *formatting* or *interpolation* operator. Given `format % values` (where *format* is a string), `%` conversion specifications in *format* are replaced with zero or more elements of *values*. The effect is similar to using the `sprintf()` function in the C language. For example :

```
>>> print('%s has %d quote types.' % ('Python', 2))
Python has 2 quote types.
```

Si *format* ne nécessite qu'un seul argument, *values* peut être un objet unique[5]. Si *values* est un *n*-uplet, il doit contenir exactement le nombre d'éléments spécifiés par la chaîne de format, ou un seul objet tableau de correspondances (*mapping object*, par exemple, un dictionnaire).

Un indicateur de conversion contient deux ou plusieurs caractères et comporte les éléments suivants, qui doivent apparaître dans cet ordre :

1. le caractère `'%'`, qui marque le début du marqueur ;
2. la clé de correspondance (facultative), composée d'une suite de caractères entre parenthèses (par exemple, `(somename)`) ;
3. des indications de conversion, facultatives, qui affectent le résultat de certains types de conversion ;
4. largeur minimum (facultative). Si elle vaut `'*'` (astérisque), la largeur est lue de l'élément suivant du *n*-uplet *values*, et l'objet à convertir vient après la largeur de champ minimale et la précision facultative ;
5. précision (facultatif), donnée sous la forme d'un `'.'` (point) suivi de la précision. Si la précision est `'*'` (un astérisque), la précision est lue à partir de l'élément suivant du *n*-uplet *values* et la valeur à convertir vient ensuite ;
6. modificateur de longueur (facultatif) ;
7. type de conversion.

Lorsque l'argument de droite est un dictionnaire (ou un autre type de tableau de correspondances), les marqueurs dans la chaîne *doivent* inclure une clé présente dans le dictionnaire, écrite entre parenthèses, immédiatement après le caractère `'%'`. La clé indique quelle valeur du dictionnaire doit être formatée. Par exemple :

```
>>> print('%(language)s has %(number)03d quote types.' %
...       {'language': "Python", "number": 2})
Python has 002 quote types.
```

Dans ce cas, aucune `*` ne peut se trouver dans le format (car ces `*` nécessitent une liste (accès séquentiel) de paramètres).

Les caractères indicateurs de conversion sont :

5. Pour insérer un *n*-uplet, vous devez donc donner un *n*-uplet d'un seul élément, contenant le *n*-uplet à insérer.

Option	Signification
`'#'`	La conversion utilise la « forme alternative » (définie ci-dessous).
`'0'`	Les valeurs numériques converties sont complétées de zéros.
`'-'`	La valeur convertie est ajustée à gauche (remplace la conversion `'0'` si les deux sont données).
`' '`	(une espace) Une espace doit être laissée avant un nombre positif (ou chaîne vide) produite par la conversion d'une valeur signée.
`'+'`	Un caractère de signe (`'+'` ou `'-'`) précède la valeur convertie (remplace le marqueur « espace »).

Un modificateur de longueur (`h`, `l` ou `L`) peut être présent, mais est ignoré car il est pas nécessaire pour Python, donc par exemple `%ld` est identique à `%d`.

Les types utilisables dans les conversions sont :

Conversion	Signification	Notes
`'d'`	Entier décimal signé.	
`'i'`	Entier décimal signé.	
`'o'`	Valeur octale signée.	(1)
`'u'`	Type obsolète — identique à `'d'`.	(6)
`'x'`	Hexadécimal signé (en minuscules).	(2)
`'X'`	Hexadécimal signé (capitales).	(2)
`'e'`	Floating-point exponential format (lowercase).	(3)
`'E'`	Floating-point exponential format (uppercase).	(3)
`'f'`	Floating-point decimal format.	(3)
`'F'`	Floating-point decimal format.	(3)
`'g'`	Floating-point format. Uses lowercase exponential format if exponent is less than -4 or not less than precision, decimal format otherwise.	(4)
`'G'`	Floating-point format. Uses uppercase exponential format if exponent is less than -4 or not less than precision, decimal format otherwise.	(4)
`'c'`	Un seul caractère (accepte des entiers ou une chaîne d'un seul caractère).	
`'r'`	String (convertit n'importe quel objet Python avec `repr()`).	(5)
`'s'`	String (convertit n'importe quel objet Python avec `str()`).	(5)
`'a'`	String (convertit n'importe quel objet Python en utilisant `ascii()`).	(5)
`'%'`	Aucun argument n'est converti, donne un caractère de `'%'` dans le résultat.	

Notes :

(1) La forme alternative entraîne l'insertion d'un préfixe octal (`'0o'`) avant le premier chiffre.

(2) La forme alternative entraîne l'insertion d'un préfixe `'0x'` ou `'0X'` (respectivement pour les formats `'x'` et `'X'`) avant le premier chiffre.

(3) La forme alternative implique la présence d'un point décimal, même si aucun chiffre ne le suit.

La précision détermine le nombre de chiffres après la virgule, 6 par défaut.

(4) La forme alternative implique la présence d'un point décimal et les zéros non significatifs sont conservés (ils ne le seraient pas autrement).

La précision détermine le nombre de chiffres significatifs avant et après la virgule. 6 par défaut.

(5) Si la précision est `N`, la sortie est tronquée à `N` caractères.

(6) Voir la **PEP 237**.

Puisque les chaînes Python ont une longueur explicite, les conversions `%s` ne considèrent pas `'\0'` comme la fin de la chaîne.

Modifié dans la version 3.1 : les conversions `%f` des nombres dont la valeur absolue est supérieure à `1e50` ne sont plus remplacées par des conversions `%g`.

4.10 Séquences Binaires — bytes, bytearray, vue mémoire

Les principaux types natifs pour manipuler des données binaires sont `bytes` et `bytearray`. Ils sont gérés par les `vues mémoire` qui utilisent le protocole tampon pour accéder à la mémoire d'autres objets binaires sans avoir besoin d'en faire une copie.

Le module `array` permet le stockage efficace de types basiques comme les entiers de 32 bits et les *float* double précision IEEE754.

4.10.1 Objets *bytes*

Les *bytes* sont des séquences immuables d'octets. Comme beaucoup de protocoles binaires utilisent l'ASCII, les objets *bytes* offrent plusieurs méthodes qui ne sont valables que lors de la manipulation de données ASCII et sont étroitement liés aux objets *str* dans bien d'autres aspects.

class bytes([*source*[, *encoding*[, *errors*]]])

Tout d'abord, la syntaxe des *bytes* littéraux est en grande partie la même que pour les chaînes littérales, en dehors du préfixe `b` :
— entre guillemets simples : `b'cela autorise les guillemets "doubles"'`;
— entre guillemets (anglais) : `b"cela permet aussi les guillemets 'simples'"`;
— entre guillemets triples : `b'''3 single quotes''',b"""3 double quotes"""`.
Seuls les caractères ASCII sont autorisés dans les littéraux de *bytes* (quel que soit l'encodage du code source déclaré). Toutes les valeurs au-delà de 127 doivent être écrites en utilisant une séquence d'échappement appropriée.

Comme avec les chaînes littérales, les *bytes* littéraux peuvent également utiliser un préfixe `r` pour désactiver le traitement des séquences d'échappement. Voir strings pour plus d'informations sur les différentes formes littérales de *bytes*, y compris les séquences d'échappement gérées.

Bien que les *bytes* littéraux, et leur représentation, soient basés sur du texte ASCII, les *bytes* se comportent en fait comme des séquences immuables de nombres entiers, dont les valeurs sont restreintes dans l'intervalle `0 <= x < 256` (ne pas respecter cette restriction lève une `ValueError`). C'est délibéré afin de souligner que, bien que de nombreux encodages binaires soient compatibles avec l'ASCII, et peuvent être manipulés avec des algorithmes orientés texte, ce n'est généralement pas le cas pour les données binaires arbitraires (appliquer aveuglément des algorithmes de texte sur des données binaires qui ne sont pas compatibles ASCII conduit généralement à leur corruption).

En plus des formes littérales, des objets *bytes* peuvent être créés par de nombreux moyens :
— un objet *bytes* rempli de zéros d'une longueur spécifiée : `bytes(10)`;
— un itérable d'entiers : `bytes(range(20))`;
— la copie de données binaires existantes via le protocole tampon : `bytes(obj)`.
Voir aussi la fonction native *bytes*.

Puisque 2 chiffres hexadécimaux correspondent précisément à un seul octet, les nombres hexadécimaux sont un format couramment utilisé pour décrire les données binaires. Par conséquent, le type *bytes* a une méthode de classe pour lire des données dans ce format :

classmethod fromhex(*string*)

Cette méthode de la classe `bytes` renvoie un objet *bytes*, décodant la chaîne donnée. La chaîne doit contenir deux chiffres hexadécimaux par octet, les espaces ASCII sont ignorés.

```
>>> bytes.fromhex('2Ef0 F1f2  ')
b'.\xf0\xf1\xf2'
```

Modifié dans la version 3.7 : `bytes.fromhex()` saute maintenant dans la chaîne tous les caractères ASCII d'espacement, pas seulement les espaces.

Une fonction de conversion inverse existe pour transformer un objet *bytes* en sa représentation hexadécimale.

hex([*sep*[, *bytes_per_sep*]])

Renvoie une chaîne contenant deux chiffres hexadécimaux pour chaque octet de l'instance.

```
>>> b'\xf0\xf1\xf2'.hex()
'f0f1f2'
```

Si vous voulez obtenir une chaîne hexadécimale plus facile à lire, vous pouvez spécifier le paramètre *sep* comme « caractère de séparation », à inclure dans la sortie. Par défaut, ce caractère est inséré entre chaque octet. Un second paramètre optionnel *bytes_per_sep* contrôle l'espacement. Les valeurs positives calculent la position du séparateur en partant de la droite, les valeurs négatives de la gauche.

```
>>> value = b'\xf0\xf1\xf2'
>>> value.hex('-')
'f0-f1-f2'
>>> value.hex('_', 2)
'f0_f1f2'
>>> b'UUDDLRLRAB'.hex(' ', -4)
'55554444 4c524c52 4142'
```

Ajouté dans la version 3.5.

Modifié dans la version 3.8 : `bytes.hex()` prend désormais en charge les paramètres optionnels *sep* et *bytes_per_sep* pour insérer des séparateurs entre les octets dans la sortie hexadécimale.

Comme les objets *bytes* sont des séquences d'entiers (semblables à un *n*-uplet), pour une instance de *bytes b*, `b[0]` est un entier, tandis que `b[0:1]` est un objet *bytes* de longueur 1. (Cela contraste avec les chaînes, où un indice et le découpage donnent une chaîne de longueur 1.)

La représentation des *bytes* utilise le format littéral (`b'...'`) car il est souvent plus utile que par exemple `bytes([46, 46, 46])`. Vous pouvez toujours convertir un *bytes* en liste d'entiers en utilisant `list(b)`.

4.10.2 Objets *bytearray*

Les objets `bytearray` sont l'équivalent mutable des objets `bytes`.

class bytearray([*source*[, *encoding*[, *errors*]]])

Il n'y a pas de syntaxe littérale dédiée aux *bytearray*, ils sont toujours créés en appelant le constructeur :
- créer une instance vide : `bytearray()` ;
- crée une instance remplie de zéros d'une longueur donnée : `bytearray(10)` ;
- à partir d'un itérable d'entiers : `bytearray(range(20))` ;
- copie des données binaires existantes via le protocole tampon : `bytearray(b'Hi!')`.

Comme les *bytearray* sont mutables, ils prennent en charge les opérations de séquences *mutables* en plus des opérations communes de *bytes* et *bytearray* décrites dans *Opérations sur les bytes et bytearray*.

Voir aussi la fonction native *bytearray*.

Puisque 2 chiffres hexadécimaux correspondent précisément à un octet, les nombres hexadécimaux sont un format couramment utilisé pour décrire les données binaires. Par conséquent, le type *bytearray* a une méthode de classe pour lire les données dans ce format :

classmethod fromhex(*string*)

Cette méthode de la classe `bytearray` renvoie un objet *bytearray*, décodant la chaîne donnée. La chaîne doit contenir deux chiffres hexadécimaux par octet, les caractères d'espacement ASCII sont ignorés.

```
>>> bytearray.fromhex('2Ef0 F1f2  ')
bytearray(b'.\xf0\xf1\xf2')
```

Modifié dans la version 3.7 : `bytearray.fromhex()` saute maintenant tous les caractères d'espacement ASCII dans la chaîne, pas seulement les espaces.

Une fonction de conversion inverse existe pour transformer un objet *bytearray* en sa représentation hexadécimale.

hex([*sep*[, *bytes_per_sep*]])

Renvoie une chaîne contenant deux chiffres hexadécimaux pour chaque octet de l'instance.

```
>>> bytearray(b'\xf0\xf1\xf2').hex()
'f0f1f2'
```

Ajouté dans la version 3.5.

Modifié dans la version 3.8 : similaire à `bytes.hex()`, `bytearray.hex()` prend désormais en charge les paramètres optionnels *sep* et *bytes_per_sep* pour insérer des séparateurs entre les octets dans la sortie hexadécimale.

resize(*size*)

Resize the `bytearray` to contain *size* bytes. *size* must be greater than or equal to 0.

If the `bytearray` needs to shrink, bytes beyond *size* are truncated.

If the `bytearray` needs to grow, all new bytes, those beyond *size*, will be set to null bytes.

This is equivalent to :

```
>>> def resize(ba, size):
...     if len(ba) > size:
...         del ba[size:]
...     else:
...         ba += b'\0' * (size - len(ba))
```

Examples :

```
>>> shrink = bytearray(b'abc')
>>> shrink.resize(1)
>>> (shrink, len(shrink))
(bytearray(b'a'), 1)
>>> grow = bytearray(b'abc')
>>> grow.resize(5)
>>> (grow, len(grow))
(bytearray(b'abc\x00\x00'), 5)
```

Ajouté dans la version 3.14.

Comme les *bytearray* sont des séquences d'entiers (semblables à une liste), pour un objet *bytearray b*, `b[0]` est un entier, tandis que `b[0:1]` est un objet *bytearray* de longueur 1. (Ceci contraste avec les chaînes de texte, où l'indice et le découpage produisent une chaîne de longueur 1)

La représentation des objets *bytearray* utilise le format littéral des *bytes* (`bytearray(b'...')`) car il est souvent plus utile que par exemple `bytearray([46, 46, 46])`. Vous pouvez toujours convertir un objet *bytearray* en une liste de nombres entiers en utilisant `list(b)`.

4.10.3 Opérations sur les *bytes* et *bytearray*

bytes et *bytearray* prennent en charge les opérations *communes* des séquences. Ils interagissent non seulement avec des opérandes de même type, mais aussi avec les *objets octet-compatibles*. En raison de cette flexibilité, ils peuvent être mélangés librement dans des opérations sans provoquer d'erreurs. Cependant, le type du résultat peut dépendre de l'ordre des opérandes.

Note

les méthodes sur les *bytes* et les *bytearray* n'acceptent pas les chaînes comme arguments, tout comme les méthodes sur les chaînes n'acceptent pas les *bytes* comme arguments. Par exemple, vous devez écrire :

```
a = "abc"
b = a.replace("a", "f")
```

et :

```
a = b"abc"
b = a.replace(b"a", b"f")
```

Quelques opérations de *bytes* et *bytesarray* supposent l'utilisation de formats binaires compatibles ASCII, et donc doivent être évités lorsque vous travaillez avec des données binaires arbitraires. Ces restrictions sont couvertes ci-dessous.

> **Note**
>
> utiliser ces opérations basées sur l'ASCII pour manipuler des données binaires qui ne sont pas au format ASCII peut les corrompre.

Les méthodes suivantes sur les *bytes* et *bytearray* peuvent être utilisées avec des données binaires arbitraires.

`bytes.count(sub[, start[, end]])`

`bytearray.count(sub[, start[, end]])`

Renvoie le nombre d'occurrences qui ne se chevauchent pas de la sous-séquence *sub* dans l'intervalle [*start*, *end*]. Les arguments facultatifs *start* et *end* sont interprétés comme dans la notation des découpages.

La sous-séquence à rechercher peut être un quelconque *objet octet-compatible* ou un nombre entier compris entre 0 et 255.

Si *sub* est vide, renvoie le nombre de tranches vides entre les caractères de début et de fin, ce qui correspond à la longueur de l'objet bytes plus un.

Modifié dans la version 3.3 : accepte aussi un nombre entier compris entre 0 et 255 comme sous-séquence.

`bytes.removeprefix(prefix, /)`

`bytearray.removeprefix(prefix, /)`

Si les données binaires commencent par la chaîne *prefix*, renvoie `bytes[len(prefix):]`. Sinon, renvoie une copie des données binaires d'origine :

```
>>> b'TestHook'.removeprefix(b'Test')
b'Hook'
>>> b'BaseTestCase'.removeprefix(b'Test')
b'BaseTestCase'
```

Le *prefix* peut être n'importe quel *objet octet-compatible*.

> **Note**
>
> la version *bytearray* de cette méthode *ne modifie pas* les octets, elle produit toujours un nouvel objet, même si aucune modification n'a été effectuée.

Ajouté dans la version 3.9.

`bytes.removesuffix(suffix, /)`

`bytearray.removesuffix(suffix, /)`

Si les données binaires terminent par la chaîne *suffix*, renvoie `bytes[:-len(suffix)]`. Sinon, renvoie une copie des données binaires d'origine :

```
>>> b'MiscTests'.removesuffix(b'Tests')
b'Misc'
>>> b'TmpDirMixin'.removesuffix(b'Tests')
b'TmpDirMixin'
```

Le *suffix* peut être n'importe quel *objet octet-compatible*.

> **Note**
>
> la version *bytearray* de cette méthode *ne modifie pas* les octets, elle produit toujours un nouvel objet, même si aucune modification n'a été effectuée.

Ajouté dans la version 3.9.

`bytes.decode`(*encoding='utf-8'*, *errors='strict'*)

`bytearray.decode`(*encoding='utf-8'*, *errors='strict'*)

Renvoie la chaine d'octets décodée en instance de `str`.

encoding vaut par défaut `utf-8` ; pour une liste des encodages possibles, voir la section *Standard Encodings*.

errors détermine la manière dont sont traitées les erreurs. Sa valeur par défaut est `'strict'`, ce qui signifie que les erreurs d'encodage lèvent une `UnicodeError`. Les autres valeurs possibles sont `'ignore'`, `'replace'` et tout autre nom enregistré *via* `codecs.register_error()`, voir la section *Gestionnaires d'erreurs* pour les détails.

Pour des raisons de performances, la valeur de *errors* n'est pas vérifiée à moins qu'une erreur de décodage ne se produise réellement, que le *mode développeur* ne soit activé ou que Python ait été compilé en mode débogage.

> **Note**
>
> passer l'argument *encoding* à `str` permet de décoder tout *objet octet-compatible* directement, sans avoir besoin d'utiliser un `bytes` ou `bytearray` temporaire.

Modifié dans la version 3.1 : gère les arguments nommés.

Modifié dans la version 3.9 : les valeurs de *errors* sont maintenant vérifiées en *mode de développement* et en mode de débogage.

`bytes.endswith`(*suffix*[, *start*[, *end*]])

`bytearray.endswith`(*suffix*[, *start*[, *end*]])

Renvoie `True` si les octets se terminent par *suffix*, sinon `False`. *suffix* peut aussi être un *n*-uplet de suffixes à rechercher. Avec l'argument optionnel *start*, la recherche se fait à partir de cette position. Avec l'argument optionnel *end*, la comparaison s'arrête à cette position.

Les suffixes à rechercher peuvent être n'importe quel *objet octet-compatible*.

`bytes.find`(*sub*[, *start*[, *end*]])

`bytearray.find`(*sub*[, *start*[, *end*]])

Renvoie la première position où le *sub* se trouve dans les données, de telle sorte que *sub* soit contenue dans `s[start:end]`. Les arguments facultatifs *start* et *end* sont interprétés comme dans la notation des découpages. Renvoie `-1` si *sub* n'est pas trouvé.

La sous-séquence à rechercher peut être un quelconque *objet octet-compatible* ou un nombre entier compris entre 0 et 255.

> **Note**
>
> la méthode `find()` ne doit être utilisée que si vous avez besoin de connaître la position de *sub*. Pour vérifier si *sub* est présent ou non, utilisez l'opérateur `in` :
>
> ```
> >>> b'Py' in b'Python'
> True
> ```

Modifié dans la version 3.3 : accepte aussi un nombre entier compris entre 0 et 255 comme sous-séquence.

`bytes.index`(*sub*[, *start*[, *end*]])

`bytearray.index`(*sub*[, *start*[, *end*]])

Comme `find()`, mais lève une `ValueError` lorsque la séquence est introuvable.

La sous-séquence à rechercher peut être un quelconque *objet octet-compatible* ou un nombre entier compris entre 0 et 255.

Modifié dans la version 3.3 : accepte aussi un nombre entier compris entre 0 et 255 comme sous-séquence.

`bytes.join`(*iterable*)

`bytearray.join`(*iterable*)

Renvoie un *bytes* ou *bytearray* qui est la concaténation des séquences de données binaires dans *iterable*. Une exception `TypeError` est levée si une valeur d'*iterable* n'est pas un *objet octet-compatible*, y compris pour des `str`. Le séparateur entre les éléments est le contenu du *bytes* ou du *bytearray* depuis lequel cette méthode est appelée.

static `bytes.maketrans`(*from*, *to*)

static `bytearray.maketrans`(*from*, *to*)

Cette méthode statique renvoie une table de traduction utilisable par `bytes.translate()` qui permettra de changer chaque caractère de *from* par un caractère à la même position dans *to* ; *from* et *to* doivent tous deux être des *objets octet-compatibles* et avoir la même longueur.

Ajouté dans la version 3.1.

`bytes.partition`(*sep*)

`bytearray.partition`(*sep*)

Divise la séquence à la première occurrence de *sep*, et renvoie un triplet contenant la partie précédant le séparateur, le séparateur lui-même (ou sa copie en *byterray*), et la partie suivant le séparateur. Si le séparateur n'est pas trouvé, le triplet renvoyé contient une copie de la séquence d'origine, suivi de deux *bytes* ou *bytearray* vides.

Le séparateur à rechercher peut être tout *objet octet-compatible*.

`bytes.replace`(*old*, *new*[, *count*])

`bytearray.replace`(*old*, *new*[, *count*])

Renvoie une copie de la séquence dont toutes les occurrences de la sous-séquence *old* sont remplacées par *new*. Si l'argument optionnel *count* est donné, seules les *count* premières occurrences sont remplacées.

La sous-séquence à rechercher et son remplacement peuvent être n'importe quel *objet octet-compatible*.

> **Note**
>
> la version *bytearray* de cette méthode *ne modifie pas* les octets, elle produit toujours un nouvel objet, même si aucune modification n'a été effectuée.

`bytes.rfind`(*sub*[, *start*[, *end*]])

`bytearray.rfind`(*sub*[, *start*[, *end*]])

Renvoie la plus grande position de *sub* dans la séquence, de telle sorte que *sub* soit dans `s[start:end]`. Les arguments facultatifs *start* et *end* sont interprétés comme dans la notation des découpages. Renvoie `-1` si *sub* n'est pas trouvable.

La sous-séquence à rechercher peut être un quelconque *objet octet-compatible* ou un nombre entier compris entre 0 et 255.

Modifié dans la version 3.3 : accepte aussi un nombre entier compris entre 0 et 255 comme sous-séquence.

`bytes.rindex`(*sub*[, *start*[, *end*]])

bytearray.**rindex**(*sub*[, *start*[, *end*]])

Semblable à `rfind()` mais lève une `ValueError` lorsque *sub* est introuvable.

La sous-séquence à rechercher peut être un quelconque *objet octet-compatible* ou un nombre entier compris entre 0 et 255.

Modifié dans la version 3.3 : accepte aussi un nombre entier compris entre 0 et 255 comme sous-séquence.

bytes.**rpartition**(*sep*)

bytearray.**rpartition**(*sep*)

Coupe la séquence à la dernière occurrence de *sep*, et renvoie un triplet de trois éléments contenant la partie précédant le séparateur, le séparateur lui-même (ou sa copie, un *bytearray*), et la partie suivant le séparateur. Si le séparateur n'est pas trouvé, le triplet contient deux *bytes* ou *bytesarray* vides suivi d'une copie de la séquence d'origine.

Le séparateur à rechercher peut être tout *objet octet-compatible*.

bytes.**startswith**(*prefix*[, *start*[, *end*]])

bytearray.**startswith**(*prefix*[, *start*[, *end*]])

Renvoie `True` si les données binaires commencent par le *prefix* spécifié, sinon `False`. *prefix* peut aussi être un *n*-uplet de préfixes à rechercher. Avec l'argument *start* la recherche commence à cette position. Avec l'argument *end* option, la recherche s'arrête à cette position.

Les préfixes à rechercher peuvent être n'importe quels *objets octet-compatibles*.

bytes.**translate**(*table*, /, *delete=b"*)

bytearray.**translate**(*table*, /, *delete=b"*)

Renvoie une copie du *bytes* ou *bytearray* dont tous les octets de *delete* sont supprimés, et les octets restants changés par la table de correspondance donnée, qui doit être un objet *bytes* d'une longueur de 256.

Vous pouvez utiliser la méthode `bytes.maketrans()` pour créer une table de correspondance.

Donnez `None` comme *table* pour seulement supprimer des caractères :

```
>>> b'read this short text'.translate(None, b'aeiou')
b'rd ths shrt txt'
```

Modifié dans la version 3.6 : *delete* est maintenant accepté comme argument nommé.

Les méthodes suivantes sur les *bytes* et *bytearray* supposent par défaut que les données traitées sont compatibles ASCII, mais peuvent toujours être utilisées avec des données binaires, arbitraires, en passant des arguments appropriés. Notez que toutes les méthodes de *bytearray* de cette section ne travaillent jamais sur l'objet lui-même, mais renvoient un nouvel objet.

bytes.**center**(*width*[, *fillbyte*])

bytearray.**center**(*width*[, *fillbyte*])

Renvoie une copie de l'objet centrée dans une séquence de longueur *width*. Le remplissage est fait en utilisant *fillbyte* (qui par défaut est une espace ASCII). Pour les objets `bytes`, la séquence initiale est renvoyée si *width* est inférieure ou égale à `len(s)`.

> **Note**
>
> la version *bytearray* de cette méthode *ne modifie pas* les octets, elle produit toujours un nouvel objet, même si aucune modification n'a été effectuée.

bytes.**ljust**(*width*[, *fillbyte*])

bytearray.**ljust**(*width*[, *fillbyte*])

Renvoie une copie de l'objet aligné à gauche dans une séquence de longueur *width*. Le remplissage est fait en utilisant *fillbyte* (par défaut un espace ASCII). Pour les objets `bytes`, la séquence initiale est renvoyée si *width* est inférieure ou égale à `len(s)`.

> **Note**
>
> la version *bytearray* de cette méthode *ne modifie pas* les octets, elle produit toujours un nouvel objet, même si aucune modification n'a été effectuée.

`bytes.lstrip([chars])`

`bytearray.lstrip([chars])`

Renvoie une copie de la séquence dont certains préfixes ont été supprimés. L'argument *chars* est une séquence binaire spécifiant le jeu d'octets à supprimer. Ce nom se réfère au fait de cette méthode est généralement utilisée avec des caractères ASCII. En cas d'omission ou `None`, la valeur par défaut de *chars* permet de supprimer des espaces ASCII. L'argument *chars* n'est pas un préfixe, toutes les combinaisons de ses valeurs sont supprimées :

```
>>> b'   spacious   '.lstrip()
b'spacious   '
>>> b'www.example.com'.lstrip(b'cmowz.')
b'example.com'
```

Les octets à retirer peuvent être n'importe quel *bytes-like object*. Voir `removeprefix()` pour une méthode qui supprime, au début de la séquence, la chaîne de caractères en tant que telle plutôt que l'ensemble des caractères passés en paramètre. Par exemple :

```
>>> b'Arthur: three!'.lstrip(b'Arthur: ')
b'ee!'
>>> b'Arthur: three!'.removeprefix(b'Arthur: ')
b'three!'
```

> **Note**
>
> la version *bytearray* de cette méthode *ne modifie pas* les octets, elle produit toujours un nouvel objet, même si aucune modification n'a été effectuée.

`bytes.rjust(width[, fillbyte])`

`bytearray.rjust(width[, fillbyte])`

Renvoie une copie de l'objet justifié à droite dans une séquence de longueur *width*. Le remplissage est fait en utilisant le caractère *fillbyte* (par défaut est un espace ASCII). Pour les objets `bytes`, la séquence d'origine est renvoyée si *width* est inférieure ou égale à `len(s)`.

> **Note**
>
> la version *bytearray* de cette méthode *ne modifie pas* les octets, elle produit toujours un nouvel objet, même si aucune modification n'a été effectuée.

`bytes.rsplit(sep=None, maxsplit=-1)`

`bytearray.rsplit(sep=None, maxsplit=-1)`

Divise la séquence d'octets en sous-séquences du même type, en utilisant *sep* comme séparateur. Si *maxsplit* est donné, c'est le nombre maximum de divisions qui pourront être faites, celles "à droite". Si *sep* est pas spécifié ou est `None`, toute sous-séquence composée uniquement d'espaces ASCII est un séparateur. En dehors du fait qu'il découpe par la droite, `rsplit()` se comporte comme `split()` qui est décrit en détail ci-dessous.

`bytes.rstrip([chars])`

`bytearray.rstrip([chars])`

Renvoie une copie de la séquence dont des octets finaux sont supprimés. L'argument *chars* est une séquence d'octets spécifiant le jeu de caractères à supprimer. En cas d'omission ou `None`, les espaces ASCII sont supprimées. L'argument *chars* n'est pas un suffixe : toutes les combinaisons de ses valeurs sont retirées :

```
>>> b'   spacious   '.rstrip()
b'   spacious'
>>> b'mississippi'.rstrip(b'ipz')
b'mississ'
```

Les octets à retirer peuvent être n'importe quel *bytes-like object*. Voir `removesuffix()` pour une méthode qui supprime, à la fin de la séquence, la chaîne de caractères en tant que telle plutôt que l'ensemble des caractères passés en paramètre. Par exemple :

```
>>> b'Monty Python'.rstrip(b' Python')
b'M'
>>> b'Monty Python'.removesuffix(b' Python')
b'Monty'
```

Note

la version *bytearray* de cette méthode *ne modifie pas* les octets, elle produit toujours un nouvel objet, même si aucune modification n'a été effectuée.

`bytes.`**`split`**(*sep=None*, *maxsplit=-1*)

`bytearray.`**`split`**(*sep=None*, *maxsplit=-1*)

Divise la séquence en sous-séquences du même type, en utilisant *sep* comme séparateur. Si *maxsplit* est donné, c'est le nombre maximum de divisions qui pourront être faites (la liste aura donc au plus `maxsplit+1` éléments), Si *maxsplit* n'est pas spécifié ou faut `-1`, il n'y a aucune limite au nombre de découpes (elles sont toutes effectuées).

If *sep* is given, consecutive delimiters are not grouped together and are deemed to delimit empty subsequences (for example, `b'1,,2'.split(b',')` returns `[b'1', b'', b'2']`). The *sep* argument may consist of a multibyte sequence as a single delimiter. Splitting an empty sequence with a specified separator returns `[b'']` or `[bytearray(b'')]` depending on the type of object being split. The *sep* argument may be any *bytes-like object*.

Par exemple :

```
>>> b'1,2,3'.split(b',')
[b'1', b'2', b'3']
>>> b'1,2,3'.split(b',', maxsplit=1)
[b'1', b'2,3']
>>> b'1,2,,3,'.split(b',')
[b'1', b'2', b'', b'3', b'']
>>> b'1<>2<>3<4'.split(b'<>')
[b'1', b'2', b'3<4']
```

Si *sep* n'est pas spécifié ou est `None`, un autre algorithme de découpe est appliqué : les espaces ASCII consécutifs sont considérés comme un seul séparateur, et le résultat ne contiendra pas les chaînes vides de début ou de la fin si la chaîne est préfixée ou suffixé d'espaces. Par conséquent, diviser une séquence vide ou une séquence composée d'espaces ASCII avec un séparateur `None` renvoie `[]`.

Par exemple :

```
>>> b'1 2 3'.split()
[b'1', b'2', b'3']
>>> b'1 2 3'.split(maxsplit=1)
[b'1', b'2 3']
>>> b'   1   2   3   '.split()
[b'1', b'2', b'3']
```

`bytes.`**`strip`**([*chars*])

bytearray.**strip**([*chars*])

Renvoie une copie de la séquence dont des caractères initiaux et finaux sont supprimés. L'argument *chars* est une séquence spécifiant le jeu d'octets à supprimer, le nom se réfère au fait de cette méthode est généralement utilisée avec des caractères ASCII. En cas d'omission ou `None`, les espaces ASCII sont supprimés. L'argument *chars* n'est ni un préfixe ni un suffixe, toutes les combinaisons de ses valeurs sont supprimées :

```
>>> b'   spacious   '.strip()
b'spacious'
>>> b'www.example.com'.strip(b'cmowz.')
b'example'
```

Les octets à retirer peuvent être tout *bytes-like object*.

> **Note**
>
> la version *bytearray* de cette méthode *ne modifie pas* les octets, elle produit toujours un nouvel objet, même si aucune modification n'a été effectuée.

Les méthodes suivantes sur les *bytes* et *bytearray* supposent l'utilisation d'un format binaire compatible ASCII, et donc doivent être évités lorsque vous travaillez avec des données binaires arbitraires. Notez que toutes les méthodes de *bytearray* de cette section *ne modifient pas* les octets, ils produisent de nouveaux objets.

bytes.**capitalize**()

bytearray.**capitalize**()

Renvoie une copie de la séquence dont chaque octet est interprété comme un caractère ASCII, le premier octet en capitale et le reste en minuscules. Les octets non ASCII ne sont pas modifiés.

> **Note**
>
> la version *bytearray* de cette méthode *ne modifie pas* les octets, elle produit toujours un nouvel objet, même si aucune modification n'a été effectuée.

bytes.**expandtabs**(*tabsize=8*)

bytearray.**expandtabs**(*tabsize=8*)

Renvoie une copie de la séquence où toutes les tabulations ASCII sont remplacées par un ou plusieurs espaces ASCII, en fonction de la colonne courante et de la taille de tabulation donnée. Les positions des tabulations se trouvent tous les *tabsize* caractères (8 par défaut, ce qui donne les positions de tabulations aux colonnes 0, 8, 16 et ainsi de suite). Pour travailler sur la séquence, la colonne en cours est mise à zéro et la séquence est examinée octets par octets. Si l'octet est une tabulation ASCII (`b'\t'`), un ou plusieurs espaces sont insérés au résultat jusqu'à ce que la colonne courante soit égale à la position de tabulation suivante. (Le caractère tabulation lui-même n'est pas copié.) Si l'octet courant est un saut de ligne ASCII (`b'\n'`) ou un retour chariot (`b'\r'`), il est copié et la colonne en cours est remise à zéro. Tout autre octet est copié inchangé et la colonne en cours est incrémentée de un indépendamment de la façon dont l'octet est représenté lors de l'affichage :

```
>>> b'01\t012\t0123\t01234'.expandtabs()
b'01      012     0123    01234'
>>> b'01\t012\t0123\t01234'.expandtabs(4)
b'01  012 0123    01234'
```

> **Note**
>
> la version *bytearray* de cette méthode *ne modifie pas* les octets, elle produit toujours un nouvel objet, même si aucune modification n'a été effectuée.

bytes.**isalnum**()

bytearray.**isalnum**()

Renvoie `True` si tous les caractères de la chaîne sont des caractères ASCII alphabétiques ou chiffres et que la séquence n'est pas vide, sinon `False`. Les caractères ASCII alphabétiques sont les suivants dans la séquence d'octets `b'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'` et les chiffres : `b'0123456789'`.

Par exemple :

```
>>> b'ABCabc1'.isalnum()
True
>>> b'ABC abc1'.isalnum()
False
```

bytes.**isalpha**()

bytearray.**isalpha**()

Renvoie `True` si tous les octets dans la séquence sont des caractères alphabétiques ASCII et que la séquence n'est pas vide, sinon `False`. Les caractères ASCII alphabétiques sont : `b'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

Par exemple :

```
>>> b'ABCabc'.isalpha()
True
>>> b'ABCabc1'.isalpha()
False
```

bytes.**isascii**()

bytearray.**isascii**()

Renvoie `True` si la séquence est vide, ou si tous ses octets sont des octets ASCII, renvoie `False` dans le cas contraire. Les octets ASCII dans l'intervalle `0-0x7F`.

Ajouté dans la version 3.7.

bytes.**isdigit**()

bytearray.**isdigit**()

Renvoie `True` si tous les octets de la séquence sont des chiffres ASCII et que la séquence n'est pas vide, sinon `False`. Les chiffres ASCII sont ceux dans la séquence d'octets `b'0123456789'`.

Par exemple :

```
>>> b'1234'.isdigit()
True
>>> b'1.23'.isdigit()
False
```

bytes.**islower**()

bytearray.**islower**()

Renvoie `True` s'il y a au moins un caractère ASCII minuscule dans la séquence et aucune capitale, sinon `False`.

Par exemple :

```
>>> b'hello world'.islower()
True
>>> b'Hello world'.islower()
False
```

Les caractères ASCII minuscules sont `b'abcdefghijklmnopqrstuvwxyz'`. Les capitales ASCII sont `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

bytes.**isspace**()

`bytearray.isspace()`

Renvoie `True` si tous les octets de la séquence sont des espaces ASCII et que la séquence n'est pas vide, sinon `False`. Les espèces ASCII sont `b' \t\n\r\x0b\f'` (espace, tabulation, saut de ligne, retour chariot, tabulation verticale, saut de page).

`bytes.istitle()`

`bytearray.istitle()`

Renvoie `True` si la séquence ASCII est *titlecased*, et qu'elle n'est pas vide, sinon `False`. Voir *`bytes.title()`* pour plus de détails sur la définition de *titlecase*.

Par exemple :

```
>>> b'Hello World'.istitle()
True
>>> b'Hello world'.istitle()
False
```

`bytes.isupper()`

`bytearray.isupper()`

Renvoie `True` s'il y a au moins un caractère alphabétique majuscule ASCII dans la séquence et aucun caractère ASCII minuscule, sinon `False`.

Par exemple :

```
>>> b'HELLO WORLD'.isupper()
True
>>> b'Hello world'.isupper()
False
```

Les caractères ASCII minuscules sont `b'abcdefghijklmnopqrstuvwxyz'`. Les capitales ASCII sont `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

`bytes.lower()`

`bytearray.lower()`

Renvoie une copie de la séquence dont tous les caractères ASCII en majuscules sont convertis en leur équivalent en minuscules.

Par exemple :

```
>>> b'Hello World'.lower()
b'hello world'
```

Les caractères ASCII minuscules sont `b'abcdefghijklmnopqrstuvwxyz'`. Les capitales ASCII sont `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

> **Note**
>
> la version *bytearray* de cette méthode *ne modifie pas* les octets, elle produit toujours un nouvel objet, même si aucune modification n'a été effectuée.

`bytes.splitlines(keepends=False)`

`bytearray.splitlines(keepends=False)`

Renvoie une liste des lignes de la séquence d'octets, découpant au niveau des fin de lignes ASCII. Cette méthode utilise l'approche *universal newlines* pour découper les lignes. Les fins de ligne ne sont pas inclus dans la liste des résultats, sauf si *keepends* est donné et vrai.

Par exemple :

```
>>> b'ab c\n\nde fg\rkl\r\n'.splitlines()
[b'ab c', b'', b'de fg', b'kl']
>>> b'ab c\n\nde fg\rkl\r\n'.splitlines(keepends=True)
[b'ab c\n', b'\n', b'de fg\r', b'kl\r\n']
```

Contrairement à *split()* lorsque le délimiteur *sep* est fourni, cette méthode renvoie une liste vide pour la chaîne vide, et un saut de ligne à la fin ne se traduit pas par une ligne supplémentaire :

```
>>> b"".split(b'\n'), b"Two lines\n".split(b'\n')
([b''], [b'Two lines', b''])
>>> b"".splitlines(), b"One line\n".splitlines()
([], [b'One line'])
```

bytes.**swapcase**()

bytearray.**swapcase**()

Renvoie une copie de la séquence dont tous les caractères ASCII minuscules sont convertis en majuscules et vice-versa.

Par exemple :

```
>>> b'Hello World'.swapcase()
b'hELLO wORLD'
```

Les caractères ASCII minuscules sont `b'abcdefghijklmnopqrstuvwxyz'`. Les capitales ASCII sont `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

Unlike *str.swapcase()*, it is always the case that `bin.swapcase().swapcase() == bin` for the binary versions. Case conversions are symmetrical in ASCII, even though that is not generally true for arbitrary Unicode code points.

> **Note**
>
> la version *bytearray* de cette méthode *ne modifie pas* les octets, elle produit toujours un nouvel objet, même si aucune modification n'a été effectuée.

bytes.**title**()

bytearray.**title**()

Renvoie une version *titlecased* de la séquence d'octets où les mots commencent par un caractère ASCII majuscule et les caractères restants sont en minuscules. Les octets non capitalisables ne sont pas modifiés.

Par exemple :

```
>>> b'Hello world'.title()
b'Hello World'
```

Les caractères ASCII minuscules sont `b'abcdefghijklmnopqrstuvwxyz'`. Les caractères ASCII majuscules sont `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`. Aucun autre octet n'est capitalisable.

Pour l'algorithme, la notion de mot est définie simplement et indépendamment de la langue comme un groupe de lettres consécutives. La définition fonctionne dans de nombreux contextes, mais cela signifie que les apostrophes (typiquement de la forme possessive en Anglais) forment les limites de mot, ce qui n'est pas toujours le résultat souhaité :

```
>>> b"they're bill's friends from the UK".title()
b"They're Bill'S Friends From The Uk"
```

Une solution pour contourner le problème des apostrophes peut être obtenue en utilisant des expressions rationnelles :

```
>>> import re
>>> def titlecase(s):
...     return re.sub(rb"[A-Za-z]+('[A-Za-z]+)?",
...                   lambda mo: mo.group(0)[0:1].upper() +
...                              mo.group(0)[1:].lower(),
...                   s)
...
>>> titlecase(b"they're bill's friends.")
b"They're Bill's Friends."
```

Note

la version *bytearray* de cette méthode *ne modifie pas* les octets, elle produit toujours un nouvel objet, même si aucune modification n'a été effectuée.

`bytes.upper()`

`bytearray.upper()`

Renvoie une copie de la séquence dont tous les caractères ASCII minuscules sont convertis en leur équivalent majuscule.

Par exemple :

```
>>> b'Hello World'.upper()
b'HELLO WORLD'
```

Les caractères ASCII minuscules sont `b'abcdefghijklmnopqrstuvwxyz'`. Les capitales ASCII sont `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

Note

la version *bytearray* de cette méthode *ne modifie pas* les octets, elle produit toujours un nouvel objet, même si aucune modification n'a été effectuée.

`bytes.zfill(width)`

`bytearray.zfill(width)`

Renvoie une copie de la séquence remplie par la gauche du chiffre `b'0'` pour en faire une séquence de longueur *width*. Un préfixe (`b'+'` / `b'-'`) est permis par l'insertion du caractère de remplissage *après* le caractère de signe plutôt qu'avant. Pour les objets `bytes` la séquence d'origine est renvoyée si *width* est inférieur ou égale à `len(seq)`.

Par exemple :

```
>>> b"42".zfill(5)
b'00042'
>>> b"-42".zfill(5)
b'-0042'
```

Note

la version *bytearray* de cette méthode *ne modifie pas* les octets, elle produit toujours un nouvel objet, même si aucune modification n'a été effectuée.

4.10.4 Formatage de *bytes* a la `printf`

> **Note**
>
> Les opérations de formatage décrites ici présentent une variété de bizarreries qui conduisent à un certain nombre d'erreurs classiques (typiquement, échouer à afficher des *n*-uplets ou des dictionnaires correctement). Si la valeur à afficher peut être un *n*-uplet ou un dictionnaire, mettez-le à l'intérieur d'un autre *n*-uplet.

Les objets *bytes* (`bytes` et `bytearray`) ont un unique opérateur : l'opérateur `%` (modulo). Il est aussi connu sous le nom d'opérateur de mise en forme. Avec `format % values` (où *format* est un objet *bytes*), les marqueurs de conversion `%` dans *format* sont remplacés par zéro ou plus de *values*. L'effet est similaire à la fonction `sprintf()` du langage C.

Si *format* ne nécessite qu'un seul argument, *values* peut être un objet unique.[page 63, 5] Si *values* est un *n*-uplet, il doit contenir exactement le nombre d'éléments spécifiés dans le format en *bytes*, ou un seul objet de correspondances (*mapping object*, par exemple, un dictionnaire).

Un indicateur de conversion contient deux ou plusieurs caractères et comporte les éléments suivants, qui doivent apparaître dans cet ordre :

1. le caractère `'%'`, qui marque le début du marqueur ;
2. la clé de correspondance (facultative), composée d'une suite de caractères entre parenthèses (par exemple, `(somename)`) ;
3. des indications de conversion, facultatives, qui affectent le résultat de certains types de conversion ;
4. largeur minimum (facultative). Si elle vaut `'*'` (astérisque), la largeur est lue de l'élément suivant du *n*-uplet *values*, et l'objet à convertir vient après la largeur de champ minimale et la précision facultative ;
5. précision (facultatif), donnée sous la forme d'un `'.'` (point) suivi de la précision. Si la précision est `'*'` (un astérisque), la précision est lue à partir de l'élément suivant du *n*-uplet *values* et la valeur à convertir vient ensuite ;
6. modificateur de longueur (facultatif) ;
7. type de conversion.

Lorsque l'argument de droite est un dictionnaire (ou un autre type de *mapping*), les marqueurs dans le *bytes doivent* inclure une clé présente dans le dictionnaire, écrite entre parenthèses, immédiatement après le caractère `'%'`. La clé indique quelle valeur du dictionnaire doit être formatée. Par exemple :

```
>>> print(b'%(language)s has %(number)03d quote types.' %
...       {b'language': b"Python", b"number": 2})
b'Python has 002 quote types.'
```

Dans ce cas, aucune `*` ne peut se trouver dans le format (car ces `*` nécessitent une liste (accès séquentiel) de paramètres).

Les caractères indicateurs de conversion sont :

Option	Signification
`'#'`	La conversion utilise la « forme alternative » (définie ci-dessous).
`'0'`	Les valeurs numériques converties sont complétées de zéros.
`'-'`	La valeur convertie est ajustée à gauche (remplace la conversion `'0'` si les deux sont données).
`' '`	(une espace) Une espace doit être laissée avant un nombre positif (ou chaîne vide) produite par la conversion d'une valeur signée.
`'+'`	Un caractère de signe (`'+'` ou `'-'`) précède la valeur convertie (remplace le marqueur « espace »).

Un modificateur de longueur (`h`, `l` ou `L`) peut être présent, mais est ignoré car il est pas nécessaire pour Python, donc par exemple `%ld` est identique à `%d`.

Les types utilisables dans les conversions sont :

Conversion	Signification	Notes
`'d'`	Entier décimal signé.	
`'i'`	Entier décimal signé.	
`'o'`	Valeur octale signée.	(1)
`'u'`	Type obsolète — identique à `'d'`.	(8)
`'x'`	Hexadécimal signé (en minuscules).	(2)
`'X'`	Hexadécimal signé (capitales).	(2)
`'e'`	Floating-point exponential format (lowercase).	(3)
`'E'`	Floating-point exponential format (uppercase).	(3)
`'f'`	Floating-point decimal format.	(3)
`'F'`	Floating-point decimal format.	(3)
`'g'`	Floating-point format. Uses lowercase exponential format if exponent is less than -4 or not less than precision, decimal format otherwise.	(4)
`'G'`	Floating-point format. Uses uppercase exponential format if exponent is less than -4 or not less than precision, decimal format otherwise.	(4)
`'c'`	Octet simple (Accepte un nombre entier ou un seul objet *byte*).	
`'b'`	Bytes (any object that follows the buffer protocol or has `__bytes__()`).	(5)
`'s'`	`'s'` est un alias de `'b'` et ne devrait être utilisé que pour du code Python2/3.	(6)
`'a'`	*Bytes* (convertit n'importe quel objet Python en utilisant `repr(obj).encode('ascii', 'backslashreplace)`).	(5)
`'r'`	`'r'` est un alias de `'a'` et ne devrait être utilise que dans du code Python2/3.	(7)
`'%'`	Aucun argument n'est converti, donne un caractère de `'%'` dans le résultat.	

Notes :

(1) La forme alternative entraîne l'insertion d'un préfixe octal (`'0o'`) avant le premier chiffre.

(2) La forme alternative entraîne l'insertion d'un préfixe `'0x'` ou `'0X'` (respectivement pour les formats `'x'` et `'X'`) avant le premier chiffre.

(3) La forme alternative implique la présence d'un point décimal, même si aucun chiffre ne le suit.

La précision détermine le nombre de chiffres après la virgule, 6 par défaut.

(4) La forme alternative implique la présence d'un point décimal et les zéros non significatifs sont conservés (ils ne le seraient pas autrement).

La précision détermine le nombre de chiffres significatifs avant et après la virgule. 6 par défaut.

(5) Si la précision est `N`, la sortie est tronquée à `N` caractères.

(6) `b'%s'` est obsolète, mais ne sera pas retiré des version 3.x.

(7) `b'%r'` est obsolète mais ne sera pas retiré dans Python 3.x.

(8) Voir la **PEP 237**.

Note

la version *bytearray* de cette méthode *ne modifie pas* les octets, elle produit toujours un nouvel objet, même si aucune modification n'a été effectuée.

Voir aussi

PEP 461 -- Ajout du formatage via % aux *bytes* et *bytesarray*

Ajouté dans la version 3.5.

4.10.5 Vues mémoire

Les *vues mémoire* permettent à du code Python d'accéder sans copie aux données internes d'un objet prenant en charge le protocole tampon.

class memoryview(*object*)

Crée une *vue mémoire* faisant référence à *object*. *object* doit savoir gérer le protocole tampon. *bytes* et *bytearray* sont des classes natives prenant en charge le protocole tampon.

Une *vue mémoire* a la notion d'*élément*, qui est l'unité de mémoire atomique gérée par l'objet *object* d'origine. Pour de nombreux types simples comme *bytes* et *bytearray*, l'élément est l'octet, mais pour d'autres types tels que *array.array* les éléments peuvent être plus grands.

`len(view)` is equal to the length of *tolist*, which is the nested list representation of the view. If `view.ndim = 1`, this is equal to the number of elements in the view.

Modifié dans la version 3.12 : If `view.ndim == 0`, `len(view)` now raises *TypeError* instead of returning 1.

The *itemsize* attribute will give you the number of bytes in a single element.

Une *vue mémoire* autorise le découpage et l'indiçage de ses données. Découper sur une dimension donne une sous-vue :

```
>>> v = memoryview(b'abcefg')
>>> v[1]
98
>>> v[-1]
103
>>> v[1:4]
<memory at 0x7f3ddc9f4350>
>>> bytes(v[1:4])
b'bce'
```

Si le *format* est un des formats natif du module *struct*, indexer avec un nombre entier ou un *n*-uplet de nombres entiers est aussi autorisé et renvoie un seul *element* du bon type. Les vues mémoire unidimensionnelles peuvent être indexées avec un nombre entier ou un *n*-uplet d'un entier. Les *memoryview* multi-dimensionnelles peuvent être indexées avec des *ndim*-uplets où *ndim* est le nombre de dimensions. Les *memoryviews* à zéro dimension peuvent être indexées avec un *n*-uplet vide.

Voici un exemple avec un autre format que *byte* :

```
>>> import array
>>> a = array.array('l', [-11111111, 22222222, -33333333, 44444444])
>>> m = memoryview(a)
>>> m[0]
-11111111
>>> m[-1]
44444444
>>> m[::2].tolist()
[-11111111, -33333333]
```

Si l'objet sous-jacent est accessible en écriture, la vue mémoire prend en charge les assignations de tranches unidimensionnelles. Redimensionner n'est cependant pas autorisé :

```
>>> data = bytearray(b'abcefg')
>>> v = memoryview(data)
>>> v.readonly
False
>>> v[0] = ord(b'z')
>>> data
```

(suite sur la page suivante)

(suite de la page précédente)

```
bytearray(b'zbcefg')
>>> v[1:4] = b'123'
>>> data
bytearray(b'z123fg')
>>> v[2:3] = b'spam'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: memoryview assignment: lvalue and rvalue have different␣
↪structures
>>> v[2:6] = b'spam'
>>> data
bytearray(b'z1spam')
```

Les vues mémoire unidimensionnelles de *hachables* (lecture seule) avec les formats 'B', 'b', ou 'c' sont aussi hachables. La fonction de hachage est définie telle que `hash(m) == hash(m.tobytes())` :

```
>>> v = memoryview(b'abcefg')
>>> hash(v) == hash(b'abcefg')
True
>>> hash(v[2:4]) == hash(b'ce')
True
>>> hash(v[::-2]) == hash(b'abcefg'[::-2])
True
```

Modifié dans la version 3.3 : Les vues mémoire unidimensionnelles peuvent aussi être découpées. Les vues mémoire unidimensionnelles avec les formats 'B', 'b' ou 'c' sont maintenant *hachables*.

Modifié dans la version 3.4 : *memoryview* est maintenant enregistrée automatiquement avec `collections.abc.Sequence`

Modifié dans la version 3.5 : les vues mémoire peuvent maintenant être indicées par un *n*-uplet d'entiers.

Modifié dans la version 3.14 : memoryview is now a *generic type*.

La classe `vue mémoire` dispose de plusieurs méthodes :

__eq__(*exporter*)

Une vue mémoire et un *exporter* de la **PEP 3118** sont égaux si leurs formes sont équivalentes et si toutes les valeurs correspondantes sont égales, les formats respectifs des opérandes étant interprétés en utilisant la syntaxe de `struct`.

Pour le sous-ensemble des formats de `struct` pris en charge par `tolist()`, `v` et `w` sont égaux si `v.tolist() ==w.tolist()` :

```
>>> import array
>>> a = array.array('I', [1, 2, 3, 4, 5])
>>> b = array.array('d', [1.0, 2.0, 3.0, 4.0, 5.0])
>>> c = array.array('b', [5, 3, 1])
>>> x = memoryview(a)
>>> y = memoryview(b)
>>> x == a == y == b
True
>>> x.tolist() == a.tolist() == y.tolist() == b.tolist()
True
>>> z = y[::-2]
>>> z == c
True
>>> z.tolist() == c.tolist()
True
```

Si l'un des formats n'est pas géré par le module de `struct`, les objets sont toujours considérés différents (même si les formats et les valeurs contenues sont identiques) :

```
>>> from ctypes import BigEndianStructure, c_long
>>> class BEPoint(BigEndianStructure):
...     _fields_ = [("x", c_long), ("y", c_long)]
...
>>> point = BEPoint(100, 200)
>>> a = memoryview(point)
>>> b = memoryview(point)
>>> a == point
False
>>> a == b
False
```

Note that, as with floating-point numbers, `v is w` does *not* imply `v == w` for memoryview objects.

Modifié dans la version 3.3 : les versions précédentes comparaient la mémoire brute sans tenir compte du format de l'objet ni de sa structure logique.

tobytes(*order='C'*)

Renvoie les données de la vue mémoire sous forme de *bytes*. Cela équivaut à appeler le constructeur `bytes` sur la vue mémoire.

```
>>> m = memoryview(b"abc")
>>> m.tobytes()
b'abc'
>>> bytes(m)
b'abc'
```

Pour les tableaux non contigus le résultat est égal à la représentation en liste aplatie dont tous les éléments sont convertis en octets. `tobytes()` prend en charge toutes les chaînes de format, y compris celles qui ne sont pas connues du module `struct`.

Ajouté dans la version 3.8 : *order* peut être `'C'`, `'F'` ou `'A'`. Lorsque *order* est `'C'` ou `'F'`, les données du tableau original sont converties en ordre C ou Fortran. Pour les vues contiguës, `'A'` renvoie une copie exacte de la mémoire physique. En particulier, l'ordre Fortran en mémoire est conservé. Pour les vues non contiguës, les données sont d'abord converties en C. `order=None` est identique à `order='C'`.

hex([*sep*[, *bytes_per_sep*]])

Renvoie une chaîne contenant deux chiffres hexadécimaux pour chaque octet de la mémoire.

```
>>> m = memoryview(b"abc")
>>> m.hex()
'616263'
```

Ajouté dans la version 3.5.

Modifié dans la version 3.8 : similaire à `bytes.hex()`, `memoryview.hex()` prend désormais en charge les paramètres optionnels *sep* et *bytes_per_sep* pour insérer des séparateurs entre les octets dans la sortie hexadécimale.

tolist()

Renvoie les données de la mémoire sous la forme d'une liste d'éléments.

```
>>> memoryview(b'abc').tolist()
[97, 98, 99]
>>> import array
>>> a = array.array('d', [1.1, 2.2, 3.3])
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> m = memoryview(a)
>>> m.tolist()
[1.1, 2.2, 3.3]
```

Modifié dans la version 3.3 : `tolist()` prend désormais en charge tous les formats d'un caractère du module `struct` ainsi que des représentations multidimensionnelles.

toreadonly()

Renvoie une version en lecture seule de l'objet *memoryview*. L'objet original *memoryview* est inchangé.

```
>>> m = memoryview(bytearray(b'abc'))
>>> mm = m.toreadonly()
>>> mm.tolist()
[97, 98, 99]
>>> mm[0] = 42
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot modify read-only memory
>>> m[0] = 43
>>> mm.tolist()
[43, 98, 99]
```

Ajouté dans la version 3.8.

release()

Libère le tampon sous-jacent exposé par l'objet *memoryview*. Beaucoup d'objets ont des comportements spécifiques lorsqu'ils sont liés à une vue (par exemple, un `bytearray` refusera temporairement de se faire redimensionner). Par conséquent, appeler *release()* peut être pratique pour lever ces restrictions (et libérer des ressources liées) aussi tôt que possible.

Après le premier appel de cette méthode, toute nouvelle opération sur la vue mémoire lève une `ValueError` (sauf `release()` elle-même qui peut être appelée plusieurs fois) :

```
>>> m = memoryview(b'abc')
>>> m.release()
>>> m[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operation forbidden on released memoryview object
```

Le protocole de gestion de contexte peut être utilisé pour obtenir un effet similaire, via l'instruction `with` :

```
>>> with memoryview(b'abc') as m:
...     m[0]
...
97
>>> m[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operation forbidden on released memoryview object
```

Ajouté dans la version 3.2.

cast(*format*[, *shape*])

Change le format ou la forme d'une vue mémoire. Par défaut *shape* vaut `[byte_length//new_itemsize]`, ce qui signifie que la vue résultante n'a qu'une dimension. La valeur renvoyée est une nouvelle vue mémoire, mais le tampon sous-jacent lui-même n'est pas copié.

Les changements de format pris en charge sont « une dimension vers C-*contiguous* » et « *C-contiguous* vers une dimension ».

The destination format is restricted to a single element native format in `struct` syntax. One of the formats must be a byte format ('B', 'b' or 'c'). The byte length of the result must be the same as the original length. Note that all byte lengths may depend on the operating system.

Transformer un *1D/long* en *1D/unsigned bytes* :

```
>>> import array
>>> a = array.array('l', [1,2,3])
>>> x = memoryview(a)
>>> x.format
'l'
>>> x.itemsize
8
>>> len(x)
3
>>> x.nbytes
24
>>> y = x.cast('B')
>>> y.format
'B'
>>> y.itemsize
1
>>> len(y)
24
>>> y.nbytes
24
```

Transformer un *1D/unsigned bytes* en *1D/char* :

```
>>> b = bytearray(b'zyz')
>>> x = memoryview(b)
>>> x[0] = b'a'
Traceback (most recent call last):
  ...
TypeError: memoryview: invalid type for format 'B'
>>> y = x.cast('c')
>>> y[0] = b'a'
>>> b
bytearray(b'ayz')
```

Transformer un *1D/bytes* en *3D/ints* en *1D/signed char* :

```
>>> import struct
>>> buf = struct.pack("i"*12, *list(range(12)))
>>> x = memoryview(buf)
>>> y = x.cast('i', shape=[2,2,3])
>>> y.tolist()
[[[0, 1, 2], [3, 4, 5]], [[6, 7, 8], [9, 10, 11]]]
>>> y.format
'i'
>>> y.itemsize
4
>>> len(y)
2
>>> y.nbytes
48
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> z = y.cast('b')
>>> z.format
'b'
>>> z.itemsize
1
>>> len(z)
48
>>> z.nbytes
48
```

Transformer un *1D/unsigned char* en *2D/unsigned long* :

```
>>> buf = struct.pack("L"*6, *list(range(6)))
>>> x = memoryview(buf)
>>> y = x.cast('L', shape=[2,3])
>>> len(y)
2
>>> y.nbytes
48
>>> y.tolist()
[[0, 1, 2], [3, 4, 5]]
```

Ajouté dans la version 3.3.

Modifié dans la version 3.5 : le format de la source n'est plus restreint lors de la transformation vers une vue d'octets.

count(*value*, /)

Count the number of occurrences of *value*.

Ajouté dans la version 3.14.

index(*value*, *start=0*, *stop=sys.maxsize*, /)

Return the index of the first occurrence of *value* (at or after index *start* and before index *stop*).

Raises a `ValueError` if *value* cannot be found.

Ajouté dans la version 3.14.

Plusieurs attributs en lecture seule sont également disponibles :

obj

L'objet sous-jacent de la vue mémoire :

```
>>> b  = bytearray(b'xyz')
>>> m = memoryview(b)
>>> m.obj is b
True
```

Ajouté dans la version 3.3.

nbytes

`nbytes == product(shape) * itemsize == len(m.tobytes())`. C'est l'espace que la liste occuperait en octets, dans une représentation contiguë. Ce n'est pas nécessairement égal à `len(m)` :

```
>>> import array
>>> a = array.array('i', [1,2,3,4,5])
>>> m = memoryview(a)
>>> len(m)
5
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> m.nbytes
20
>>> y = m[::2]
>>> len(y)
3
>>> y.nbytes
12
>>> len(y.tobytes())
12
```

Tableaux multidimensionnels :

```
>>> import struct
>>> buf = struct.pack("d"*12, *[1.5*x for x in range(12)])
>>> x = memoryview(buf)
>>> y = x.cast('d', shape=[3,4])
>>> y.tolist()
[[0.0, 1.5, 3.0, 4.5], [6.0, 7.5, 9.0, 10.5], [12.0, 13.5, 15.0, 16.5]]
>>> len(y)
3
>>> y.nbytes
96
```

Ajouté dans la version 3.3.

readonly

Booléen indiquant si la mémoire est en lecture seule.

format

Chaîne contenant le format (dans le style de *struct*) pour chaque élément de la vue. Une vue mémoire peut être créée depuis des exportateurs de formats arbitraires, mais certaines méthodes (comme *tolist()*) sont limitées aux formats natifs à un seul élément.

Modifié dans la version 3.3 : le format `'B'` est maintenant traité selon la syntaxe du module *struct*. Cela signifie que `memoryview(b'abc')[0] == b'abc'[0] == 97`.

itemsize

Taille en octets de chaque élément de la vue mémoire :

```
>>> import array, struct
>>> m = memoryview(array.array('H', [32000, 32001, 32002]))
>>> m.itemsize
2
>>> m[0]
32000
>>> struct.calcsize('H') == m.itemsize
True
```

ndim

Nombre de dimensions du tableau multi-dimensionnel pointé par la vue mémoire.

shape

ndim-uplet d'entiers donnant la forme du tableau à N dimensions pointé par la vue mémoire.

Modifié dans la version 3.3 : le *n*-uplet est vide au lieu de `None` lorsque *ndim* = 0.

strides

ndim-uplet d'entiers donnant la taille en octets permettant d'accéder à chaque élément pour chaque dimension du tableau.

Modifié dans la version 3.3 : le *n*-uplet est vide au lieu de `None` lorsque *ndim* = 0.

suboffsets

Détail de l'implémentation des *PIL-style arrays*. La valeur n'est donnée qu'à titre d'information.

c_contiguous

Booléen indiquant si la mémoire est C-*contiguë*.

Ajouté dans la version 3.3.

f_contiguous

Booléen indiquant si la mémoire est Fortran-*contiguë*.

Ajouté dans la version 3.3.

contiguous

Booléen indiquant si la mémoire est *contiguë*.

Ajouté dans la version 3.3.

4.11 Types d'ensembles — `set`, `frozenset`

Un ensemble (objet *set*) est une collection non triée d'objets *hachables* distincts. Les utilisations classiques sont le test d'appartenance, la déduplication d'une séquence, ou le calcul d'opérations mathématiques telles que l'intersection, l'union, la différence, ou la différence symétrique. (Pour les autres conteneurs, voir les classes natives `dict`, `liste`, et `n-uplet`, ainsi que le module `collections`.)

Comme pour les autres collections, les ensembles gèrent `x in set`, `len(set)`, et `for x in set`. En tant que collection non triée, les ensembles n'enregistrent pas la position des éléments ou leur ordre d'insertion. En conséquence, les ensembles n'autorisent ni l'indexation, ni le découpage, ou tout autre comportement de séquence.

Il existe actuellement deux types natifs pour les ensembles, `set` et `frozenset`. Le type `set` est mutable — son contenu peut changer en utilisant des méthodes comme `add()` et `remove()`. Puisqu'il est mutable, il n'a pas de valeur de hachage et ne peut donc pas être utilisé ni comme clé de dictionnaire ni comme élément d'un autre ensemble. Le type `frozenset` est immuable et *hachable* — son contenu ne peut être modifié après sa création, il peut ainsi être utilisé comme clé de dictionnaire ou élément d'un autre ensemble.

Des *sets* (mais pas des *frozensets*) peuvent être créés par une liste d'éléments séparés par des virgules et entre accolades, par exemple `{'jack', 'sjoerd'}`, en plus du constructeur de la classe `set`.

Les constructeurs des deux classes fonctionnent de la même manière :

class set([*iterable*])

class frozenset([*iterable*])

Renvoie un nouvel ensemble (*set* ou *frozenset*) dont les éléments viennent d'*iterable*. Les éléments d'un *set* doivent être *hachables*. Pour représenter des ensembles d'ensembles, les ensembles intérieurs doivent être des `frozenset`. Si *iterable* n'est pas spécifié, un nouvel ensemble vide est renvoyé.

Les ensembles peuvent être construits de différentes manières :
— en utilisant une liste d'éléments séparés par des virgules entre accolades : `{'jack', 'sjoerd'}`;
— en utilisant un ensemble en compréhension : `{c for c in 'abracadabra' if c not in 'abc'}`;
— en utilisant le constructeur du type : `set()`, `set('foobar')`, `set(['a', 'b', 'foo'])`.
Les instances de `set` et `frozenset` fournissent les opérations suivantes :

len(s)

Renvoie le nombre d'éléments dans l'ensemble *s* (cardinalité de *s*).

x in s

Test d'appartenance de *x* dans *s*.

x not in s

Test de non-appartenance de *x* dans *s*.

isdisjoint(*other*)

Renvoie `True` si l'ensemble n'a aucun élément en commun avec *other*. Les ensembles sont disjoints si et seulement si leur intersection est un ensemble vide.

issubset(*other*)

set <= other

Teste si tous les éléments de l'ensemble sont dans *other*.

set < other

Teste si l'ensemble est un sous-ensemble propre de *other*, c'est-à-dire `set <= other and set != other`.

issuperset(*other*)

set >= other

Teste si tous les éléments de *other* sont dans l'ensemble.

set > other

Teste si l'ensemble est un sur-ensemble propre de *other*, c'est-à-dire, `set >= other and set != other`.

union(**others*)

set | other | ...

Renvoie un nouvel ensemble dont les éléments viennent de l'ensemble et de tous les autres (*others*).

intersection(**others*)

set & other & ...

Renvoie un nouvel ensemble dont les éléments sont communs à l'ensemble et à tous les autres (*others*).

difference(**others*)

set - other - ...

Renvoie un nouvel ensemble dont les éléments sont dans l'ensemble mais ne sont dans aucun des autres.

symmetric_difference(*other*)

set ^ other

Renvoie un nouvel ensemble dont les éléments sont soit dans l'ensemble, soit dans *other*, mais pas dans les deux.

copy()

Renvoie une copie superficielle du dictionnaire.

Remarque : les méthodes *union()*, *intersection()*, *difference()*, *symmetric_difference()*, *issubset()* et *issuperset()* acceptent n'importe quel itérable comme argument, contrairement aux opérateurs équivalents qui n'acceptent que des ensembles. Il est donc préférable d'éviter les constructions comme `set('abc') & 'cbs'`, sources typiques d'erreurs, en faveur d'une construction plus lisible : `set('abc').intersection('cbs')`.

Les classes *set* et *frozenset* gèrent les comparaisons d'ensemble à ensemble. Deux ensembles sont égaux si et seulement si chaque élément de chaque ensemble est contenu dans l'autre (autrement dit que chaque ensemble est un sous-ensemble de l'autre). Un ensemble est plus petit qu'un autre ensemble si et seulement si le premier est un sous-ensemble propre du second (un sous-ensemble, mais pas égal). Un ensemble est plus grand qu'un autre ensemble si et seulement si le premier est un sur-ensemble propre du second (est un sur-ensemble mais n'est pas égal).

Les instances de *set* se comparent aux instances de *frozenset* en fonction de leurs membres. Par exemple, `set('abc') == frozenset('abc')` envoie `True`, ainsi que `set('abc') in set([frozenset('abc')])`.

Les tests de sous-ensemble et d'égalité ne se généralisent pas pour former une fonction donnant un ordre total. Par exemple, deux ensembles disjoints non vides ne sont ni égaux et ni des sous-ensembles l'un de l'autre, donc toutes ces comparaisons donnent `False` : `a<b`, `a==b` et `a>b`.

Puisque les ensembles ne définissent qu'un ordre partiel (par leurs relations de sous-ensembles), la sortie de la méthode `list.sort()` n'est pas définie pour des listes d'ensembles.

Les éléments des ensembles, comme les clés de dictionnaires, doivent être *hachables*.

Les opérations binaires mélangeant des instances de `set` et `frozenset` renvoient le type de la première opérande. Par exemple, `frozenset('ab') | set('bc')` renvoie une instance de `frozenset`.

La table suivante liste les opérations disponibles pour les `set` mais qui ne s'appliquent pas aux instances de `frozenset` :

update(**others*)

set |= other | ...

Met à jour l'ensemble, ajoutant les éléments de tous les autres (*others*).

intersection_update(**others*)

set &= other & ...

Met à jour l'ensemble, ne gardant que les éléments trouvés dans tous les autres.

difference_update(**others*)

set -= other | ...

Met à jour l'ensemble, retirant les éléments trouvés dans les autres.

symmetric_difference_update(*other*)

set ^= other

Met à jour l'ensemble, ne gardant que les éléments trouvés dans un des deux ensembles mais pas dans les deux.

add(*elem*)

Ajoute l'élément *elem* à l'ensemble.

remove(*elem*)

Retire l'élément *elem* de l'ensemble. Lève une exception `KeyError` si *elem* n'est pas dans l'ensemble.

discard(*elem*)

Retire l'élément *elem* de l'ensemble s'il y est.

pop()

Retire et renvoie un élément arbitraire de l'ensemble. Lève une exception `KeyError` si l'ensemble est vide.

clear()

Supprime tous les éléments de l'ensemble.

Notez que les versions n'utilisant pas la syntaxe d'opérateur des méthodes `update()`, `intersection_update()`, `difference_update()`, et `symmetric_difference_update()` acceptent n'importe quel itérable comme argument.

Note, the *elem* argument to the `__contains__()`, `remove()`, and `discard()` methods may be a set. To support searching for an equivalent frozenset, a temporary one is created from *elem*.

4.12 Les types de correspondances — `dict`

Un objet *tableau de correspondances* (*mapping*) fait correspondre des valeurs *hachables* à des objets arbitraires. Les tableaux de correspondances sont des objets mutables. Il n'existe pour le moment qu'un type de tableau de correspondances standard, le *dictionary*. (Pour les autres conteneurs, voir les types natifs `liste`, `ensemble` et `n-uplet`, ainsi que le module `collections`.)

Les clés d'un dictionnaire sont *presque* des données arbitraires. Les valeurs qui ne sont pas *hachables*, c'est-à-dire les valeurs contenant des listes, dictionnaires ou autres types mutables (qui sont comparés à l'aide de leurs valeurs plutôt que par l'identité de l'objet) ne peuvent pas être utilisées comme clés. Des valeurs qui sont considérées égales lors d'une comparaison (comme `1`, `1.0` et `True`) peuvent être utilisées pour obtenir la même entrée d'un dictionnaire.

class dict(***kwargs*)

class dict(*mapping*, ***kwargs*)

class dict(*iterable*, ***kwargs*)

Renvoie un nouveau dictionnaire initialisé à partir d'un argument positionnel optionnel, et un ensemble (vide ou non) d'arguments nommés.

Les dictionnaires peuvent être construits de différentes manières :

— en utilisant une liste de paires `clé: valeur` séparées par des virgules entre accolades : `{'jack': 4098, 'sjoerd': 4127}` ou `{4098: 'jack', 4127: 'sjoerd'}`;

— en utilisant un dictionnaire en compréhension : `{}`, `{x: x ** 2 for x in range(10)}`;

— en utilisant le constructeur du type : `dict()`, `dict([('foo', 100), ('bar', 200)])`, `dict(foo=100, bar=200)`.

If no positional argument is given, an empty dictionary is created. If a positional argument is given and it defines a `keys()` method, a dictionary is created by calling `__getitem__()` on the argument with each returned key from the method. Otherwise, the positional argument must be an *iterable* object. Each item in the iterable must itself be an iterable with exactly two elements. The first element of each item becomes a key in the new dictionary, and the second element the corresponding value. If a key occurs more than once, the last value for that key becomes the corresponding value in the new dictionary.

Si des arguments nommés sont donnés, ils sont ajoutés au dictionnaire créé depuis l'argument positionnel. Si une clé est déjà présente, la valeur de l'argument nommé remplace la valeur reçue par l'argument positionnel.

Typiquement, les exemples suivants renvoient tous un dictionnaire valant `{"one": 1, "two": 2, "three": 3}`:

```
>>> a = dict(one=1, two=2, three=3)
>>> b = {'one': 1, 'two': 2, 'three': 3}
>>> c = dict(zip(['one', 'two', 'three'], [1, 2, 3]))
>>> d = dict([('two', 2), ('one', 1), ('three', 3)])
>>> e = dict({'three': 3, 'one': 1, 'two': 2})
>>> f = dict({'one': 1, 'three': 3}, two=2)
>>> a == b == c == d == e == f
True
```

Fournir les arguments nommés comme dans le premier exemple ne fonctionne que pour des clés qui sont des identifiants valides en Python. Dans les autres cas, toutes les clés valides sont utilisables.

Voici les opérations gérées par les dictionnaires (par conséquent, d'autres types de tableaux de correspondances devraient les gérer aussi) :

list(d)

Renvoie une liste de toutes les clés utilisées dans le dictionnaire *d*.

len(d)

Renvoie le nombre d'éléments dans le dictionnaire *d*.

d[key]

Renvoie l'élément de *d* dont la clé est *key*. Lève une exception `KeyError` si *key* n'est pas dans le dictionnaire.

Si une sous-classe de *dict* définit une méthode `__missing__()` et que *key* manque, l'opération `d[key]` appelle cette méthode avec la clé *key* en argument. L'opération `d[key]` renvoie la valeur, ou lève l'exception renvoyée ou levée par l'appel à `__missing__(key)`. Aucune autre opération ni méthode n'appelle `__missing__()`. Si `__missing__()` n'est pas définie, une exception `KeyError` est levée. `__missing__()` doit être une méthode ; ça ne peut pas être une variable d'instance :

```
>>> class Counter(dict):
...     def __missing__(self, key):
...         return 0
...
>>> c = Counter()
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> c['red']
0
>>> c['red'] += 1
>>> c['red']
1
```

L'exemple ci-dessus montre une partie de l'implémentation de `collections.Counter`. `collections.defaultdict` implémente aussi `__missing__`.

d[key] = value

Assigne `d[key]` à *value*.

del d[key]

Supprime `d[key]` de *d*. Lève une exception `KeyError` si *key* n'est pas dans le dictionnaire.

key in d

Renvoie `True` si *d* a la clé *key*, sinon `False`.

key not in d

Équivalent à `not key in d`.

iter(d)

Renvoie un itérateur sur les clés du dictionnaire. C'est un raccourci pour `iter(d.keys())`.

clear()

Supprime tous les éléments du dictionnaire.

copy()

Renvoie une copie superficielle du dictionnaire.

classmethod fromkeys(*iterable*, *value=None*, /)

Crée un nouveau dictionnaire avec les clés de *iterable* et les valeurs à *value*.

`fromkeys()` est une méthode de classe qui renvoie un nouveau dictionnaire. *value* vaut par défaut `None`. Toutes les valeurs se réfèrent à une seule instance, donc il n'est généralement pas logique que *value* soit un objet mutable comme une liste vide. Pour avoir des valeurs distinctes, utilisez plutôt une compréhension de dictionnaire.

get(*key*, *default=None*)

Renvoie la valeur de *key* si *key* est dans le dictionnaire, sinon *default*. Si *default* n'est pas donné, il vaut `None` par défaut, de manière à ce que cette méthode ne lève jamais `KeyError`.

items()

Renvoie une nouvelle vue des éléments du dictionnaire (paires de `(key, value)`). Voir la *documentation des vues*.

keys()

Renvoie une nouvelle vue des clés du dictionnaire. Voir la *documentation des vues*.

pop(*key*[, *default*])

Si *key* est dans le dictionnaire elle est supprimée et sa valeur est renvoyée, sinon renvoie *default*. Si *default* n'est pas donné et que *key* n'est pas dans le dictionnaire, une `KeyError` est levée.

popitem()

Supprime et renvoie une paire `(key, value)` du dictionnaire. Les paires sont renvoyées dans un ordre LIFO (Last In - First Out c.-à-d. dernier entré, premier sorti).

`popitem()` est pratique pour itérer un dictionnaire de manière destructive, comme souvent dans les algorithmes sur les ensembles. Si le dictionnaire est vide, appeler `popitem()` lève une `KeyError`.

Modifié dans la version 3.7 : l'ordre « dernier entré, premier sorti » (LIFO) est désormais assuré. Dans les versions précédentes, `popitem()` renvoyait une paire clé-valeur arbitraire.

reversed(d)

Renvoie un itérateur inversé sur les clés du dictionnaire. C'est un raccourci pour `reversed(d.keys())`.

Ajouté dans la version 3.8.

setdefault(*key*, *default=None*)

Si *key* est dans le dictionnaire, sa valeur est renvoyée. Sinon, insère *key* avec comme valeur *default* et renvoie *default*. *default* vaut `None` par défaut.

update([*other*])

Met à jour le dictionnaire avec les paires de clé-valeur d'*other*, écrasant les clés existantes. Renvoie `None`.

update() accepts either another object with a `keys()` method (in which case `__getitem__()` is called with every key returned from the method) or an iterable of key/value pairs (as tuples or other iterables of length two). If keyword arguments are specified, the dictionary is then updated with those key/value pairs : `d.update(red=1, blue=2)`.

values()

Renvoie une nouvelle vue des valeurs du dictionnaire. Voir la *documentation des vues*.

Une comparaison d'égalité entre une vue de `dict.values()` et une autre renvoie toujours `False`. Cela s'applique aussi lorsque l'on compare `dict.values()` à lui-même :

```
>>> d = {'a': 1}
>>> d.values() == d.values()
False
```

d | other

Crée un nouveau dictionnaire avec les clés et les valeurs fusionnées de *d* et *other*, qui doivent tous deux être des dictionnaires. Les valeurs de *other* sont prioritaires lorsque *d* et *other* partagent des clés.

Ajouté dans la version 3.9.

d |= other

Met à jour le dictionnaire *d* avec les clés et les valeurs de *other*, qui peut être soit un *tableau de correspondances* soit un *itérable* de paires clé-valeur. Les valeurs de *other* sont prioritaires lorsque *d* et *other* partagent des clés.

Ajouté dans la version 3.9.

Deux dictionnaires sont égaux si et seulement s'ils ont les mêmes paires de clé-valeur (`(key, value)`, peu importe leur ordre). Les comparaisons d'ordre (<, <=, >=, >) lèvent une *TypeError*.

Les dictionnaires préservent l'ordre des insertions. Notez que modifier une clé n'affecte pas l'ordre. Les clés ajoutées après un effacement sont insérées à la fin.

```
>>> d = {"one": 1, "two": 2, "three": 3, "four": 4}
>>> d
{'one': 1, 'two': 2, 'three': 3, 'four': 4}
>>> list(d)
['one', 'two', 'three', 'four']
>>> list(d.values())
[1, 2, 3, 4]
>>> d["one"] = 42
>>> d
{'one': 42, 'two': 2, 'three': 3, 'four': 4}
>>> del d["two"]
>>> d["two"] = None
>>> d
{'one': 42, 'three': 3, 'four': 4, 'two': None}
```

Modifié dans la version 3.7 : l'ordre d'un dictionnaire est toujours l'ordre des insertions. Ce comportement était un détail d'implémentation de CPython depuis la version 3.6.

Les dictionnaires et les vues de dictionnaires sont réversibles.

```
>>> d = {"one": 1, "two": 2, "three": 3, "four": 4}
>>> d
{'one': 1, 'two': 2, 'three': 3, 'four': 4}
>>> list(reversed(d))
['four', 'three', 'two', 'one']
>>> list(reversed(d.values()))
[4, 3, 2, 1]
>>> list(reversed(d.items()))
[('four', 4), ('three', 3), ('two', 2), ('one', 1)]
```

Modifié dans la version 3.8 : les dictionnaires sont maintenant réversibles.

Voir aussi

`types.MappingProxyType` peut être utilisé pour créer une vue en lecture seule d'un `dict`.

4.12.1 Les vues de dictionnaires

Les objets renvoyés par `dict.keys()`, `dict.values()` et `dict.items()` sont des *vues*. Ils fournissent une vue dynamique des éléments du dictionnaire, ce qui signifie que si le dictionnaire change, la vue reflète ces changements.

Les vues de dictionnaires peuvent être itérées et ainsi renvoyer les données du dictionnaire ; elles gèrent aussi les tests d'appartenance :

`len(dictview)`

Renvoie le nombre d'entrées du dictionnaire.

`iter(dictview)`

Renvoie un itérateur sur les clés, les valeurs ou les éléments (représentés par des paires `(clé, valeur)`) du dictionnaire.

Les clés et les valeurs sont itérées dans l'ordre de leur insertion. Ceci permet la création de paires de `(key, value)` en utilisant `zip()` : `pairs = zip(d.values(), d.keys())`. Un autre moyen de construire la même liste est `pairs = [(v, k) for (k, v) in d.items()]`.

Parcourir des vues tout en ajoutant ou supprimant des entrées dans un dictionnaire peut lever une `RuntimeError` ou ne pas fournir toutes les entrées.

Modifié dans la version 3.7 : l'ordre d'un dictionnaire est toujours l'ordre des insertions.

`x in dictview`

Renvoie `True` si *x* est dans les clés, les valeurs ou les éléments du dictionnaire sous-jacent (dans le dernier cas, *x* doit être une paire `(key, value)`).

`reversed(dictview)`

Renvoie un itérateur inversé sur les clés, les valeurs ou les éléments du dictionnaire. La vue est itérée dans l'ordre inverse d'insertion.

Modifié dans la version 3.8 : les vues de dictionnaires sont dorénavant réversibles.

`dictview.mapping`

Renvoie une `types.MappingProxyType` qui encapsule le dictionnaire original auquel la vue se réfère.

Ajouté dans la version 3.10.

Keys views are set-like since their entries are unique and *hashable*. Items views also have set-like operations since the (key, value) pairs are unique and the keys are hashable. If all values in an items view are hashable as well, then the items view can interoperate with other sets. (Values views are not treated as set-like since the entries are generally not unique.) For set-like views, all of the operations defined for the abstract base class `collections.abc.Set` are

available (for example, ==, <, or ^). While using set operators, set-like views accept any iterable as the other operand, unlike sets which only accept sets as the input.

Voici un exemple d'utilisation de vue de dictionnaire :

```
>>> dishes = {'eggs': 2, 'sausage': 1, 'bacon': 1, 'spam': 500}
>>> keys = dishes.keys()
>>> values = dishes.values()

>>> # iteration
>>> n = 0
>>> for val in values:
...     n += val
...
>>> print(n)
504

>>> # keys and values are iterated over in the same order (insertion order)
>>> list(keys)
['eggs', 'sausage', 'bacon', 'spam']
>>> list(values)
[2, 1, 1, 500]

>>> # view objects are dynamic and reflect dict changes
>>> del dishes['eggs']
>>> del dishes['sausage']
>>> list(keys)
['bacon', 'spam']

>>> # set operations
>>> keys & {'eggs', 'bacon', 'salad'}
{'bacon'}
>>> keys ^ {'sausage', 'juice'} == {'juice', 'sausage', 'bacon', 'spam'}
True
>>> keys | ['juice', 'juice', 'juice'] == {'bacon', 'spam', 'juice'}
True

>>> # get back a read-only proxy for the original dictionary
>>> values.mapping
mappingproxy({'bacon': 1, 'spam': 500})
>>> values.mapping['spam']
500
```

4.13 Le type gestionnaire de contexte

L'instruction Python `with` permet de créer des contextes d'exécution à l'aide de gestionnaires de contextes. L'implémentation est réalisée via deux méthodes permettant aux classes définies par l'utilisateur de définir un contexte d'exécution, dans lequel on entre avant l'exécution du corps de l'instruction et que l'on quitte lorsque l'instruction se termine :

`contextmanager.__enter__()`

Entre dans le contexte d'exécution, soit se renvoyant lui-même, soit en renvoyant un autre objet en lien avec ce contexte. La valeur renvoyée par cette méthode est liée à l'identifiant donné au `as` de l'instruction `with` utilisant ce gestionnaire de contexte.

Un exemple de gestionnaire de contexte se renvoyant lui-même est l'*objet fichier*. Les objets fichiers se renvoient eux-mêmes depuis `__enter__()` pour ainsi permettre à `open()` d'être utilisée comme expression dans une instruction `with`.

Un exemple de gestionnaire de contexte renvoyant un objet connexe est celui renvoyé par `decimal.localcontext()`. Ces gestionnaires remplacent le contexte décimal courant par une copie de l'original, copie qui est renvoyée. Ça permet de changer le contexte courant dans le corps du `with` sans affecter le code en dehors de l'instruction `with`.

`contextmanager.__exit__`(*exc_type*, *exc_val*, *exc_tb*)

Sort du contexte et renvoie un booléen indiquant si une exception est survenue et doit être supprimée. Si une exception est survenue lors de l'exécution du corps de l'instruction `with`, les arguments contiennent le type de l'exception, sa valeur et la trace de la pile (*traceback*). Sinon les trois arguments valent `None`.

L'instruction `with` inhibe l'exception si cette méthode renvoie la valeur vraie, l'exécution continuant ainsi à l'instruction suivant immédiatement l'instruction `with`. Sinon, l'exception continue de se propager après la fin de cette méthode. Les exceptions se produisant pendant l'exécution de cette méthode remplacent toute exception qui s'est produite dans le corps du `with`.

L'exception reçue ne doit jamais être relancée explicitement, cette méthode devrait plutôt renvoyer une valeur fausse pour indiquer que son exécution s'est terminée avec succès et qu'elle ne veut pas supprimer l'exception. Ceci permet au code de gestion du contexte de comprendre si une méthode `__exit__()` a échoué.

Python définit plusieurs gestionnaires de contexte pour faciliter la synchronisation des fils d'exécution, la fermeture des fichiers ou d'autres objets, et la configuration du contexte arithmétique décimal. Ces types spécifiques ne sont pas traités différemment, ils respectent simplement le protocole de gestion du contexte. Voir les exemples dans la documentation du module `contextlib`.

Les *générateurs* Python et le décorateur `contextlib.contextmanager` permettent d'implémenter simplement ces protocoles. Si un générateur est décoré avec `contextlib.contextmanager`, il renvoie un gestionnaire de contexte implémentant les méthodes `__enter__()` et `__exit__()`, plutôt que l'itérateur produit par un générateur non décoré.

Notez qu'il n'y a pas d'emplacement spécifique pour ces méthodes dans la structure de type pour les objets Python dans l'API Python/C. Les types souhaitant définir ces méthodes doivent les fournir comme une méthode accessible en Python. Comparé au coût de la mise en place du contexte d'exécution, le coût d'un accès au dictionnaire d'une classe unique est négligeable.

4.14 Types d'annotation de type — Alias générique, Union

Les principaux types natifs pour l'*annotation de types* sont l'*Alias générique* et l'*Union*.

4.14.1 Type Alias générique

Les objets `GenericAlias` sont généralement créés en indiçant une classe. Ils sont le plus souvent utilisés avec des classes conteneurs, telles que `list` ou `dict`. Par exemple, `list[int]` est un objet GenericAlias créé en indiçant la classe `list` avec l'argument `int`. Les objets `GenericAlias` sont principalement destinés à être utilisés en tant qu'*annotations de types*.

> **Note**
>
> il n'est généralement possible d'indicer une classe que si la classe implémente la méthode spéciale `__class_getitem__()`.

Un objet `GenericAlias` agit comme un mandataire pour un *type générique*, en implémentant des *types génériques pouvant recevoir des paramètres*.

Pour une classe conteneur, les arguments fournis comme indices de la classe indiquent les types des éléments que l'objet peut contenir. Par exemple, `set[bytes]` peut être utilisé dans les annotations de type pour signifier un `ensemble` dans lequel tous les éléments sont de type `bytes`.

Pour une classe qui définit `__class_getitem__()` mais n'est pas un conteneur, les arguments fournis comme indices de la classe indiquent souvent les types de retour d'une ou plusieurs méthodes définies sur un objet. Par

exemple, *re* (expressions rationnelles) peut être utilisé à la fois sur le type de données *str* et sur le type de données *bytes* :

— si `x = re.search('foo', 'foo')`, `x` est un objet *re.Match* où les valeurs de retour de `x.group(0)` et `x[0]` sont toutes les deux de type *str*. Nous pouvons représenter ce type d'objet dans des annotations de type avec le `GenericAlias` `re.Match[str]` ;

— si `y = re.search(b'bar', b'bar')`, (notez le `b` pour *bytes*), `y` est également une instance de `re.Match`, mais les valeurs de retour de `y.group(0)` et `y[0]` sont toutes les deux de type *bytes*. Dans les annotations de type, nous représenterons cette variété d'objets *re.Match* par `re.Match[bytes]`.

Les objets `GenericAlias` sont des instances de la classe *types.GenericAlias*, qui peut également être utilisée pour créer directement des objets `GenericAlias`.

T[X, Y, ...]

Crée un `GenericAlias` représentant un type `T` paramétré par les types *X*, *Y* et plus selon le `T` utilisé. Par exemple, pour une fonction attendant une *list* contenant des éléments *float* :

```
def average(values: list[float]) -> float:
    return sum(values) / len(values)
```

Un autre exemple peut être un objet *tableau de correspondances*, utilisant un *dict*, qui est un type générique attendant deux paramètres de type représentant le type clé et le type valeur. Dans cet exemple, la fonction attend un `dict` avec des clés de type *str* et des valeurs de type *int* :

```
def send_post_request(url: str, body: dict[str, int]) -> None:
    ...
```

Les fonctions natives *isinstance()* et *issubclass()* n'acceptent pas les types `GenericAlias` pour leur second argument :

```
>>> isinstance([1, 2], list[str])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: isinstance() argument 2 cannot be a parameterized generic
```

Lors de l'exécution, Python ne regarde pas les *annotations de type*. Cela vaut pour les types génériques et les types qu'on leur passe en paramètres. Lors de la création d'un objet conteneur à partir d'un `GenericAlias`, les types des éléments du conteneur ne sont pas vérifiés. Par exemple, le code suivant est déconseillé, mais s'exécute sans erreur :

```
>>> t = list[str]
>>> t([1, 2, 3])
[1, 2, 3]
```

De plus, les types génériques pouvant recevoir des paramètres effacent les paramètres de type lors de la création d'objet :

```
>>> t = list[str]
>>> type(t)
<class 'types.GenericAlias'>

>>> l = t()
>>> type(l)
<class 'list'>
```

Appeler *repr()* ou *str()* sur un type générique affiche le type pouvant recevoir des paramètres :

```
>>> repr(list[int])
'list[int]'

>>> str(list[int])
'list[int]'
```

La méthode `__getitem__()` des conteneurs génériques lève une exception pour interdire les erreurs telles que `dict[str][str]` :

```
>>> dict[str][str]
Traceback (most recent call last):
  ...
TypeError: dict[str] is not a generic class
```

Cependant, de telles expressions sont valides lorsque *des variables de type* sont utilisées. L'indice doit avoir autant d'éléments qu'il y a d'éléments variables de type dans l'objet `GenericAlias` *`__args__`*.

```
>>> from typing import TypeVar
>>> Y = TypeVar('Y')
>>> dict[str, Y][int]
dict[str, int]
```

Classes génériques standards

Les classes suivantes de la bibliothèque standard prennent en charge les types génériques pouvant accepter des paramètres. Cette liste est non exhaustive.

- *`tuple`*
- *`list`*
- *`dict`*
- *`set`*
- *`frozenset`*
- *`type`*
- *`collections.deque`*
- *`collections.defaultdict`*
- *`collections.OrderedDict`*
- *`collections.Counter`*
- *`collections.ChainMap`*
- *`collections.abc.Awaitable`*
- *`collections.abc.Coroutine`*
- *`collections.abc.AsyncIterable`*
- *`collections.abc.AsyncIterator`*
- *`collections.abc.AsyncGenerator`*
- *`collections.abc.Iterable`*
- *`collections.abc.Iterator`*
- *`collections.abc.Generator`*
- *`collections.abc.Reversible`*
- *`collections.abc.Container`*
- *`collections.abc.Collection`*
- *`collections.abc.Callable`*
- *`collections.abc.Set`*
- *`collections.abc.MutableSet`*
- *`collections.abc.Mapping`*
- *`collections.abc.MutableMapping`*
- *`collections.abc.Sequence`*
- *`collections.abc.MutableSequence`*
- *`collections.abc.MappingView`*
- *`collections.abc.KeysView`*
- *`collections.abc.ItemsView`*
- *`collections.abc.ValuesView`*
- *`contextlib.AbstractContextManager`*
- *`contextlib.AbstractAsyncContextManager`*
- *`dataclasses.Field`*
- *`functools.cached_property`*
- *`functools.partialmethod`*

— os.PathLike
— queue.LifoQueue
— queue.Queue
— queue.PriorityQueue
— queue.SimpleQueue
— *re.Pattern*
— *re.Match*
— shelve.BsdDbShelf
— shelve.DbfilenameShelf
— shelve.Shelf
— types.MappingProxyType
— weakref.WeakKeyDictionary
— weakref.WeakMethod
— weakref.WeakSet
— weakref.WeakValueDictionary

Attributs spéciaux des alias génériques

Tous les types génériques pouvant accepter des paramètres implémentent des attributs spéciaux en lecture seule.

genericalias.**__origin__**

Cet attribut pointe vers la classe générique sans paramètres :

```
>>> list[int].__origin__
<class 'list'>
```

genericalias.**__args__**

Cet attribut est un *n-uplet* (éventuellement de longueur 1) de types génériques passés à la `__class_getitem__()` d'origine de la classe générique :

```
>>> dict[str, list[int]].__args__
(<class 'str'>, list[int])
```

genericalias.**__parameters__**

Cet attribut est un *n*-uplet calculé paresseusement (éventuellement vide) de variables de type (chaque type n'est mentionné qu'une seule fois) trouvées dans `__args__` :

```
>>> from typing import TypeVar

>>> T = TypeVar('T')
>>> list[T].__parameters__
(~T,)
```

Note

un objet `GenericAlias` avec des paramètres *typing.ParamSpec* peut ne pas avoir de `__parameters__` corrects après substitution car *typing.ParamSpec* est principalement destiné à la vérification de type statique.

genericalias.**__unpacked__**

Booléen qui est vrai si l'alias a été décompressé à l'aide de l'opérateur `*` (voir *TypeVarTuple*).

Ajouté dans la version 3.11.

Voir aussi

PEP 484 - Indications des types (page en anglais)
Présentation du cadre Python pour les annotations de type.

PEP 585 – Types génériques d'indication de type dans les conteneurs standard (page en anglais)
Présentation de la possibilité de paramétrer nativement les classes de la bibliothèque standard, à condition qu'elles implémentent la méthode de classe spéciale `__class_getitem__()`.

***Génériques, génériques personnalisés* et `typing.Generic`**
Documentation sur la façon d'implémenter des classes génériques qui peuvent être paramétrées à l'exécution et comprises par les vérificateurs de type statiques.

Ajouté dans la version 3.9.

4.14.2 Type Union

Un objet *union* contient la valeur de l'opération `|` (OU bit à bit) sur plusieurs *objets types*. Ces types sont principalement destinés aux *annotations de types*. L'expression d'union de types permet une syntaxe plus claire pour les indications de type par rapport à `typing.Union`.

`X | Y | ...`

Définit un objet *union* qui contient les types *X*, *Y*, etc. `X | Y` signifie X ou Y. Cela équivaut à `typing.Union[X, Y]`. Par exemple, la fonction suivante attend un argument de type `int` ou `float` :

```
def square(number: int | float) -> int | float:
    return number ** 2
```

> **Note**
> The `|` operand cannot be used at runtime to define unions where one or more members is a forward reference. For example, `int | "Foo"`, where `"Foo"` is a reference to a class not yet defined, will fail at runtime. For unions which include forward references, present the whole expression as a string, e.g. `"int | Foo"`.

`union_object == other`

Les objets *union* peuvent être testés pour savoir s'ils sont égaux à d'autres objets *union*. Plus en détails :

— Les unions d'unions sont aplaties :

```
(int | str) | float == int | str | float
```

— Les types redondants sont supprimés :

```
int | str | int == int | str
```

— Lors de la comparaison d'unions, l'ordre est ignoré :

```
int | str == str | int
```

— Il est compatible avec `typing.Union` :

```
int | str == typing.Union[int, str]
```

— Les types optionnels peuvent être orthographiés comme une union avec `None` :

```
str | None == typing.Optional[str]
```

`isinstance(obj, union_object)`

`issubclass(obj, union_object)`

Les appels à `isinstance()` et `issubclass()` sont également pris en charge avec un objet *union* :

```
>>> isinstance("", int | str)
True
```

However, *parameterized generics* in union objects cannot be checked :

```
>>> isinstance(1, int | list[int])  # short-circuit evaluation
True
>>> isinstance([1], int | list[int])
Traceback (most recent call last):
  ...
TypeError: isinstance() argument 2 cannot be a parameterized generic
```

Le type indiqué à l'utilisateur pour l'objet *union* est accessible par `types.UnionType` et utilisé pour les vérifications `isinstance()`. Un objet ne peut pas être instancié à partir du type :

```
>>> import types
>>> isinstance(int | str, types.UnionType)
True
>>> types.UnionType()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot create 'types.UnionType' instances
```

Note

The `__or__()` method for type objects was added to support the syntax `X | Y`. If a metaclass implements `__or__()`, the Union may override it :

```
>>> class M(type):
...     def __or__(self, other):
...         return "Hello"
...
>>> class C(metaclass=M):
...     pass
...
>>> C | int
'Hello'
>>> int | C
int | C
```

Voir aussi

PEP 604 – PEP proposant la syntaxe `X | Y` et le type *Union*.

Ajouté dans la version 3.10.

4.15 Autres types natifs

L'interpréteur gère aussi d'autres types d'objets, la plupart ne gèrent cependant qu'une ou deux opérations.

4.15.1 Modules

La seule opération spéciale sur un module est l'accès à ses attributs : `m.name`, où *m* est un module et *name* donne accès un nom défini dans la table des symboles de *m*. Il est possible d'assigner un attribut de module. (Notez que l'instruction `import` n'est pas strictement une opération sur un objet module. `import foo` ne nécessite pas qu'un objet module nommé *foo* existe, il nécessite cependant une *définition* (externe) d'un module nommé *foo* quelque part.)

Un attribut spécial à chaque module est `__dict__`. C'est le dictionnaire contenant la table des symboles du module. Modifier ce dictionnaire change la table des symboles du module, mais assigner directement `__dict__` n'est pas possible (vous pouvez écrire `m.__dict__['a'] = 1`, qui donne `1` comme valeur pour `m.a`, mais vous ne pouvez pas écrire `m.__dict__ = {}`). Modifier `__dict__` directement n'est pas recommandé.

Les modules natifs de l'interpréteur sont affichés comme `<module 'sys' (built-in)>`. S'ils sont chargés depuis un fichier, ils sont affichés sous la forme `<module 'os' from '/usr/local/lib/pythonX.Y/os.pyc'>`.

4.15.2 Les classes et instances de classes

Voir objects et class.

4.15.3 Fonctions

Les objets fonctions sont créés par les définitions de fonctions. La seule opération applicable à un objet fonction est de l'appeler : `func(argument-list)`.

Il existe en fait deux catégories d'objets fonctions : les fonctions natives et les fonctions définies par l'utilisateur. Les deux gèrent les mêmes opérations (l'appel à la fonction), mais leur implémentation est différente, d'où les deux types distincts.

Voir function pour plus d'information.

4.15.4 Méthodes

Methods are functions that are called using the attribute notation. There are two flavors : built-in methods (such as `append()` on lists) and class instance method. Built-in methods are described with the types that support them.

If you access a method (a function defined in a class namespace) through an instance, you get a special object : a *bound method* (also called instance method) object. When called, it will add the `self` argument to the argument list. Bound methods have two special read-only attributes : `m.__self__` is the object on which the method operates, and `m.__func__` is the function implementing the method. Calling `m(arg-1, arg-2, ..., arg-n)` is completely equivalent to calling `m.__func__(m.__self__, arg-1, arg-2, ..., arg-n)`.

Like function objects, bound method objects support getting arbitrary attributes. However, since method attributes are actually stored on the underlying function object (`method.__func__`), setting method attributes on bound methods is disallowed. Attempting to set an attribute on a method results in an `AttributeError` being raised. In order to set a method attribute, you need to explicitly set it on the underlying function object :

```
>>> class C:
...     def method(self):
...         pass
...
>>> c = C()
>>> c.method.whoami = 'my name is method'  # can't set on the method
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'method' object has no attribute 'whoami'
>>> c.method.__func__.whoami = 'my name is method'
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> c.method.whoami
'my name is method'
```

Voir instance-methods pour plus d'information.

4.15.5 Objets code

Code objects are used by the implementation to represent "pseudo-compiled" executable Python code such as a function body. They differ from function objects because they don't contain a reference to their global execution environment. Code objects are returned by the built-in `compile()` function and can be extracted from function objects through their `__code__` attribute. See also the `code` module.

L'accès à `__code__` déclenche un *événement d'audit* `object.__getattr__` avec les arguments `obj` et `"__code__"`.

Les objets code peuvent être exécutés ou évalués en les passant (au lieu d'une chaîne contenant du code) aux fonctions natives `exec()` ou `eval()`.

Voir types pour plus d'information.

4.15.6 Objets type

Les objets types représentent les différents types d'objets. Le type d'un objet est obtenu via la fonction native `type()`. Il n'existe aucune opération spéciale sur les types. Le module standard `types` définit les noms de tous les types natifs.

Les types sont affichés comme suit : `<class 'int'>`.

4.15.7 L'objet Null

Cet objet est renvoyé par les fonctions ne renvoyant pas explicitement une valeur. Il ne gère aucune opération spéciale. Il existe exactement un seul objet *null* nommé `None` (c'est un nom natif). `type(None)()` produit ce singleton.

Il s'écrit `None`.

4.15.8 L'objet points de suspension (ou ellipse)

Cet objet est utilisé classiquement lors des découpes (voir slicings). Il ne gère aucune opération spéciale. Il n'y a qu'un seul objet *points de suspension*, nommé `Ellipsis` (un nom natif). `type(Ellipsis)()` produit le *singleton* `Ellipsis`.

Il s'écrit `Ellipsis` ou `...`.

4.15.9 L'objet *NotImplemented*

Cet objet est renvoyé depuis des comparaisons ou des opérations binaires effectuées sur des types qu'elles ne gèrent pas. Voir comparisons pour plus d'informations. Il n'y a qu'un seul objet `NotImplemented`. `type(NotImplemented)()` renvoie cette instance de singleton.

Il s'écrit `NotImplemented`.

4.15.10 Objets internes

See types for this information. It describes stack frame objects, traceback objects, and slice objects.

4.16 Attributs spéciaux

L'implémentation ajoute quelques attributs spéciaux en lecture seule à certains types, lorsque ça a du sens. Certains ne sont pas listés par la fonction native `dir()`.

`definition.__name__`
Nom de la classe, fonction, méthode, descripteur ou instance du générateur.

`definition.__qualname__`
Nom qualifié de la classe, fonction, méthode, descripteur ou instance du générateur.

Ajouté dans la version 3.3.

`definition.__module__`
The name of the module in which a class or function was defined.

`definition.__doc__`
The documentation string of a class or function, or `None` if undefined.

`definition.__type_params__`
The type parameters of generic classes, functions, and *type aliases*. For classes and functions that are not generic, this will be an empty tuple.

Ajouté dans la version 3.12.

4.17 Limitation de longueur de conversion de chaîne vers un entier

CPython a une limite globale pour la conversion entre `int` et `str` pour atténuer les attaques par déni de service. Cette limite s'applique *uniquement* aux décimaux ou autres bases de nombres qui ne sont pas des puissances de deux. Les conversions hexadécimales, octales et binaires sont illimitées. La limite peut être configurée.

Le type `int` dans CPython stocke un nombre de longueur arbitraire sous forme binaire (communément appelé « *bignum* »). Il n'existe aucun algorithme capable de convertir une chaîne en un entier binaire ou un entier binaire en une chaîne en temps linéaire, sauf si la base est une puissance de 2. Même les meilleurs algorithmes connus pour la base 10 ont une complexité sous-quadratique. La conversion d'une grande valeur telle que `int('1' * 500_000)` peut prendre plus d'une seconde sur un CPU rapide.

Limiter la taille de la conversion offre un moyen pratique d'éviter la vulnérabilité **CVE 2020-10735**.

La limite est appliquée au nombre de caractères numériques dans la chaîne d'entrée ou de sortie lorsqu'un algorithme de conversion non linéaire doit être appliqué. Les traits de soulignement et le signe ne sont pas comptés dans la limite.

Si une opération va dépasser la limite, une `ValueError` est levée :

```
>>> import sys
>>> sys.set_int_max_str_digits(4300)  # Illustrative, this is the default.
>>> _ = int('2' * 5432)
Traceback (most recent call last):
...
ValueError: Exceeds the limit (4300 digits) for integer string conversion: value␣
↪has 5432 digits; use sys.set_int_max_str_digits() to increase the limit
>>> i = int('2' * 4300)
>>> len(str(i))
4300
>>> i_squared = i*i
>>> len(str(i_squared))
Traceback (most recent call last):
...
ValueError: Exceeds the limit (4300 digits) for integer string conversion; use sys.
↪set_int_max_str_digits() to increase the limit
>>> len(hex(i_squared))
7144
>>> assert int(hex(i_squared), base=16) == i*i  # Hexadecimal is unlimited.
```

La limite par défaut est de 4 300 chiffres comme indiqué dans `sys.int_info.default_max_str_digits`. La limite la plus basse pouvant être configurée est de 640 chiffres, comme indiqué dans `sys.int_info.str_digits_check_threshold`.

Vérification :

```
>>> import sys
>>> assert sys.int_info.default_max_str_digits == 4300, sys.int_info
>>> assert sys.int_info.str_digits_check_threshold == 640, sys.int_info
>>> msg = int('578966293710682886880994035146873798396722250538762761564'
...           '9252925514383915483333812743580549779436104706260696366600'
...           '571186405732').to_bytes(53, 'big')
...
```

Ajouté dans la version 3.11.

4.17.1 API concernées

La limitation s'applique uniquement aux conversions potentiellement lentes entre *int* et *str* ou *bytes* :

- `int(string)` en base 10 (par défaut).
- `int(string, base)` pour toutes les bases qui ne sont pas des puissances de 2.
- `str(integer)`.
- `repr(integer)`.
- toute autre conversion de chaîne en base 10, par exemple `f"{integer}"`, `"{}".format(integer)` ou `b"%d" % integer`.

Les limitations ne s'appliquent pas aux fonctions avec un algorithme linéaire :

- `int(chaîne, base)` en base 2, 4, 8, 16 ou 32.
- *int.from_bytes()* et *int.to_bytes()*.
- *hex()*, *oct()*, *bin()*.
- *Mini-langage de spécification de format* pour les nombres hexadécimaux, octaux et binaires.
- *str* vers *float*.
- *str* vers *decimal.Decimal*.

4.17.2 Configuration de la limite

Avant le démarrage de Python, vous pouvez utiliser une variable d'environnement ou une option de ligne de commande d'interpréteur pour configurer la limite :

- `PYTHONINTMAXSTRDIGITS`, par exemple `PYTHONINTMAXSTRDIGITS=640 python3` pour fixer la limite à 640 chiffres ou `PYTHONINTMAXSTRDIGITS=0 python3` pour désactiver la limitation.
- `-X int_max_str_digits`, par exemple `python3 -X int_max_str_digits=640`
- *sys.flags.int_max_str_digits* contient la valeur de `PYTHONINTMAXSTRDIGITS` ou `-X int_max_str_digits`. Si la variable d'environnement et l'option `-X` sont définies toutes les deux, l'option `-X` est prioritaire. Une valeur de *-1* indique que les deux n'étaient pas définies, donc qu'une valeur de *sys.int_info.default_max_str_digits* a été utilisée lors de l'initialisation.

Depuis le code, vous pouvez inspecter la limite actuelle et en définir une nouvelle à l'aide de ces API *sys* :

- *sys.get_int_max_str_digits()* et *sys.set_int_max_str_digits()* sont un accesseur et un mutateur pour la limite relative à l'interpréteur. Les sous-interprètes possèdent leur propre limite.

Des informations sur la valeur par défaut et le minimum peuvent être trouvées dans *sys.int_info* :

- *sys.int_info.default_max_str_digits* est la limite par défaut pour la compilation.
- *sys.int_info.str_digits_check_threshold* est la valeur la plus basse acceptée pour la limite (autre que 0 qui la désactive).

Ajouté dans la version 3.11.

Prudence

fixer une limite basse *peut* entraîner des problèmes. Bien que rare, du code contenant des constantes entières en décimal dans la source qui dépasse le seuil minimum peut exister. Une conséquence de la définition de la limite est que le code source Python contenant des littéraux entiers décimaux plus longs que la limite lèvera une erreur lors de l'analyse, généralement au démarrage ou à l'importation ou même au moment de l'installation – dès qu'un `.pyc` à jour n'existe pas déjà pour le code. Une solution de contournement pour les sources qui contiennent de si grandes constantes consiste à les convertir au format hexadécimal `0x` car il n'y a pas de limite.

Testez soigneusement votre application si vous utilisez une limite basse. Assurez-vous que vos tests s'exécutent avec la limite définie au début *via* l'environnement ou l'option de ligne de commande afin qu'elle s'applique au démarrage et même lors de toute étape d'installation pouvant invoquer Python pour compiler les sources `.py` en fichiers `.pyc`.

4.17.3 Configuration recommandée

La valeur par défaut *`sys.int_info.default_max_str_digits`* devrait être raisonnable pour la plupart des applications. Si votre application nécessite une limite différente, définissez-la à partir de votre point d'entrée principal à l'aide d'un code indépendant de la version Python, car ces API ont été ajoutées dans des correctifs de sécurité des versions antérieures à 3.12.

Par exemple :

```
>>> import sys
>>> if hasattr(sys, "set_int_max_str_digits"):
...     upper_bound = 68000
...     lower_bound = 4004
...     current_limit = sys.get_int_max_str_digits()
...     if current_limit == 0 or current_limit > upper_bound:
...         sys.set_int_max_str_digits(upper_bound)
...     elif current_limit < lower_bound:
...         sys.set_int_max_str_digits(lower_bound)
```

Pour la désactiver entièrement, réglez-la à `0`.

Notes

CHAPITRE 5

Exceptions natives

En python, une exception est une instance d'une classe héritée de *BaseException*. Dans un bloc `try`, la clause `except` traite non seulement la classe d'exception qu'elle mentionne, mais aussi toutes les classes dérivées de cette classe (contrairement à ses classes mères). Deux classes qui ne sont pas liées par héritage ne sont jamais équivalentes, même si elles ont le même nom.

The built-in exceptions listed in this chapter can be generated by the interpreter or built-in functions. Except where mentioned, they have an "associated value" indicating the detailed cause of the error. This may be a string or a tuple of several items of information (e.g., an error code and a string explaining the code). The associated value is usually passed as arguments to the exception class's constructor.

Du code utilisateur peut lever des exceptions natives. Cela peut être utilisé pour tester un gestionnaire d'exception ou pour rapporter une condition d'erreur "comme si" c'était l'interpréteur qui levait cette exception ; mais attention car rien n'empêche du code utilisateur de lever une erreur inappropriée.

Les classes d'exception natives peuvent être héritées pour définir de nouvelles exceptions ; les programmeurs sont encouragés à faire dériver les nouvelles exceptions de la classe *Exception* ou d'une de ses sous-classes, et non de *BaseException*. Plus d'informations sur la définition des exceptions sont disponibles dans le Tutoriel Python au chapitre tut-userexceptions.

5.1 Contexte des exceptions

Three attributes on exception objects provide information about the context in which the exception was raised :

`BaseException.`**`__context__`**

`BaseException.`**`__cause__`**

`BaseException.`**`__suppress_context__`**

When raising a new exception while another exception is already being handled, the new exception's `__context__` attribute is automatically set to the handled exception. An exception may be handled when an `except` or `finally` clause, or a `with` statement, is used.

Ce contexte implicite d'exception peut être complété par une cause explicite en utilisant `from` avec `raise` :

```
raise new_exc from original_exc
```

The expression following `from` must be an exception or `None`. It will be set as `__cause__` on the raised exception. Setting `__cause__` also implicitly sets the `__suppress_context__` attribute to `True`, so that using `raise new_exc from None` effectively replaces the old exception with the new one for display purposes

(e.g. converting `KeyError` to `AttributeError`), while leaving the old exception available in `__context__` for introspection when debugging.

The default traceback display code shows these chained exceptions in addition to the traceback for the exception itself. An explicitly chained exception in `__cause__` is always shown when present. An implicitly chained exception in `__context__` is shown only if `__cause__` is `None` and `__suppress_context__` is false.

Dans les deux cas, l'exception elle-même est toujours affichée après toutes les exceptions enchaînées, de sorte que la dernière ligne de la trace d'appels montre toujours la dernière exception qui a été levée.

5.2 Hériter des exceptions natives

Le code de l'utilisateur peut créer des sous-classes qui dérivent d'un type d'exception. Il est recommandé de ne dériver que d'un seul type d'exception à la fois pour éviter des conflits possibles dans la façon dont les classes mères traitent l'argument `args` ainsi que des incompatibilités potentielles avec l'utilisation de la mémoire.

La majorité des exceptions natives sont implémentées en C pour des raisons d'efficacité, voir Objects/exceptions.c. L'architecture interne de certaines est telle que cela rend impossible la création de sous-classes qui dérivent de plusieurs types d'exceptions. L'agencement de la mémoire est un détail d'implémentation qui est sujet à changement d'une version de Python à l'autre, ce qui peut poser conflit dans le futur. Il est donc déconseillé de dériver de plusieurs types d'exceptions.

5.3 Classes mères

Les exceptions suivantes sont utilisées principalement en tant que classes mères pour d'autres exceptions.

exception BaseException

La classe mère pour toutes les exceptions natives. Elle n'est pas vouée à être héritée directement par des classes utilisateur (pour cela, utilisez `Exception`). Si `str()` est appelée sur une instance de cette classe, la représentation du ou des argument(s) de l'instance est retournée, ou la chaîne vide s'il n'y avait pas d'arguments.

args

Le *n*-uplet d'arguments donné au constructeur d'exception. Certaines exceptions natives (comme `OSError`) attendent un certain nombre d'arguments et attribuent une signification spéciale aux éléments de ce *n*-uplet, alors que d'autres ne sont généralement appelées qu'avec une seule chaîne de caractères rendant un message d'erreur.

with_traceback(*tb*)

Cette méthode affecte *tb* comme la nouvelle trace d'appels de l'exception et renvoie l'objet exception. Elle était utilisée de façon plus courante avant que la fonctionnalité de chaînage des exceptions de la **PEP 3134** devienne disponible. L'exemple suivant démontre comment convertir une instance de `SomeException` en une instance de `OtherException` tout en préservant la pile d'appels. Une fois l'exception levée, le cadre courant est empilé sur la trace d'appels de `OtherException`, comme cela se serait produit pour la trace d'appels de `SomeException` si on l'avait laissée se propager jusqu'à l'appelant

```
try:
    ...
except SomeException:
    tb = sys.exception().__traceback__
    raise OtherException(...).with_traceback(tb)
```

__traceback__

A writable field that holds the traceback object associated with this exception. See also : raise.

add_note(*note*)

Add the string `note` to the exception's notes which appear in the standard traceback after the exception string. A `TypeError` is raised if `note` is not a string.

Ajouté dans la version 3.11.

__notes__

A list of the notes of this exception, which were added with `add_note()`. This attribute is created when `add_note()` is called.

Ajouté dans la version 3.11.

exception Exception

Toutes les exceptions natives, qui n'entraînent pas une sortie du système dérivent de cette classe. Toutes les exceptions définies par l'utilisateur devraient également être dérivées de cette classe.

exception ArithmeticError

La classe mère pour les exceptions natives qui sont levées pour diverses erreurs arithmétiques : `OverflowError`, `ZeroDivisionError`, `FloatingPointError`.

exception BufferError

Levée lorsqu'une opération liée à un tampon ne peut pas être exécutée.

exception LookupError

La classe mère pour les exceptions qui sont levées lorsqu'une clé ou un index utilisé sur un tableau de correspondances ou une séquence est invalide : `IndexError`, `KeyError`. Peut être levée directement par `codecs.lookup()`.

5.4 Exceptions concrètes

Les exceptions suivantes sont celles qui sont habituellement levées.

exception AssertionError

Levée lorsqu'une instruction `assert` échoue.

exception AttributeError

Levée lorsqu'une référence ou une assignation d'attribut (voir attribute-references) échoue. (Lorsqu'un objet ne supporte pas du tout la référence ou l'assignation d'attribut, `TypeError` est levé.)

Les attributs `name` et `obj` peuvent être définis uniquement à l'aide d'arguments nommés passés au constructeur. Lorsqu'ils sont définis, ils représentent respectivement le nom de l'attribut pour lequel il y a eu une tentative d'accès et l'objet qui a été accédé pour cet attribut.

Modifié dans la version 3.10 : Ajout des attributs `name` et `obj`.

exception EOFError

Levée lorsque la fonction `input()` atteint une condition de fin de fichier (EOF) sans lire aucune donnée. (N.B. : les méthodes `io.IOBase.read()` et `io.IOBase.readline()` retournent une chaîne vide lorsqu'elles atteignent EOF.)

exception FloatingPointError

N'est pas utilisé pour le moment.

exception GeneratorExit

Levée lorsqu'un *generator* ou une *coroutine* est fermé, voir `generator.close()` et `coroutine.close()`. Elle hérite directement de `BaseException` au lieu de `Exception` puisqu'il ne s'agit pas techniquement d'une erreur.

exception ImportError

Levée lorsque l'instruction `import` a des problèmes pour essayer de charger un module. Également levée lorsque Python ne trouve pas un nom dans `from ... import`.

The optional *name* and *path* keyword-only arguments set the corresponding attributes :

name

The name of the module that was attempted to be imported.

path

The path to any file which triggered the exception.

Modifié dans la version 3.3 : Ajout des attributs *name* et *path*.

exception ModuleNotFoundError

Une sous-classe de *ImportError* qui est levée par `import` lorsqu'un module n'a pas pu être localisé. Elle est généralement levée quand `None` est trouvé dans *sys.modules*.

Ajouté dans la version 3.6.

exception IndexError

Levée lorsqu'un indice de séquence est hors de la plage. (Les indices de tranches (*slices*) sont tronqués silencieusement pour tomber dans la plage autorisée ; si un indice n'est pas un entier, *TypeError* est levée.)

exception KeyError

Levée lorsqu'une clef (de dictionnaire) n'est pas trouvée dans l'ensemble des clefs existantes.

exception KeyboardInterrupt

Levée lorsque l'utilisateur appuie sur la touche d'interruption (normalement `Control-C` or `Delete`). Pendant l'exécution, un contrôle des interruptions est effectué régulièrement. L'exception hérite de *BaseException* afin de ne pas être accidentellement interceptée par du code qui intercepte *Exception* et ainsi empêcher l'interpréteur de quitter.

> **Note**
>
> Attraper une *KeyboardInterrupt* demande une considération particulière. Comme elle peut être levée à des moments imprévisibles, elle peut dans certains cas laisser le programme en cours d'exécution dans un état incohérent. Il est généralement préférable de laisser *KeyboardInterrupt* arrêter le programme aussi rapidement que possible ou d'éviter catégoriquement de la lever (voir *Note on Signal Handlers and Exceptions*).

exception MemoryError

Levée lorsqu'une opération est à court de mémoire mais que la situation peut encore être rattrapée (en supprimant certains objets). La valeur associée est une chaîne de caractères indiquant quel type d'opération (interne) est à court de mémoire. À noter qu'en raison de l'architecture interne de gestion de la mémoire (la fonction `malloc()` du C), l'interpréteur peut ne pas toujours être capable de rattraper cette situation ; il lève néanmoins une exception pour qu'une pile d'appels puisse être affichée, dans le cas où un programme en cours d'exécution en était la cause.

exception NameError

Levée lorsqu'un nom local ou global n'est pas trouvé. Ceci ne s'applique qu'aux noms non qualifiés. La valeur associée est un message d'erreur qui inclut le nom qui n'a pas pu être trouvé.

L'attribut `name` doit être défini uniquement à l'aide d'un argument nommé passé au constructeur. Lorsqu'il est défini, il représente le nom de la variable pour laquelle il y a eu une tentative d'accès.

Modifié dans la version 3.10 : Ajout de l'attribut `name`.

exception NotImplementedError

Cette exception est dérivée de *RuntimeError*. Dans les classes mères définies par l'utilisateur, les méthodes abstraites devraient lever cette exception lorsqu'elles nécessitent des classes dérivées pour remplacer la méthode, ou lorsque la classe est en cours de développement pour indiquer que l'implémentation concrète doit encore être ajoutée.

> **Note**
>
> Elle ne devrait pas être utilisée pour indiquer qu'un opérateur ou qu'une méthode n'est pas destiné à être pris en charge du tout – dans ce cas, évitez de définir l'opérateur ou la méthode, ou s'il s'agit d'une sous-classe, assignez-le à *None*.

> **Note**
>
> `NotImplementedError` and `NotImplemented` are not interchangeable, even though they have similar names and purposes. See `NotImplemented` for details on when to use it.

exception OSError([*arg*])

exception OSError(*errno*, *strerror*[, *filename*[, *winerror*[, *filename2*]]])

Cette exception est levée lorsqu'une fonction système retourne une erreur liée au système, incluant les erreurs entrées-sorties telles que "fichier non trouvé" ou "disque plein" (pas pour les types d'arguments illégaux ou d'autres erreurs accidentelles).

La deuxième forme du constructeur définit les attributs correspondants, décrits ci-dessous. Les attributs par défaut sont `None` si non spécifiés. Pour la rétrocompatibilité, si trois arguments sont passés, l'attribut `args` contient seulement une paire avec les valeurs des deux premiers arguments du constructeur.

Le constructeur retourne souvent une sous-classe d'`OSError`, comme décrit dans *OS exceptions* ci-dessous. La sous-classe particulière dépend de la valeur finale d'`errno`. Ce comportement ne se produit que lors de la construction d'`OSError` directement ou via un alias, et n'est pas hérité lors du sous-classement.

errno

Code d'erreur numérique de la variable C `errno`.

winerror

Sous Windows, cela donne le code d'erreur Windows natif. L'attribut `errno` est alors une traduction approximative, en termes POSIX, de ce code d'erreur natif.

Sous Windows, si l'argument du constructeur *winerror* est un entier, l'attribut `errno` est déterminé à partir du code d'erreur Windows, et l'argument *errno* est ignoré. Sur d'autres plateformes, l'argument *winerror* est ignoré, et l'attribut `winerror` n'existe pas.

strerror

Le message d'erreur correspondant, tel que fourni par le système d'exploitation. Il est formaté par les fonctions C `perror()` sous POSIX, et `FormatMessage()` sous Windows.

filename

filename2

Pour les exceptions qui font référence à un chemin d'accès au système de fichiers (comme `open()` ou `os.unlink()`), `filename` est le nom du fichier transmis à la fonction. Pour les fonctions qui font référence à deux chemins d'accès au système de fichiers (comme `os.rename()`), `filename2` correspond au deuxième nom de fichier passé à la fonction.

Modifié dans la version 3.3 : `EnvironmentError`, `IOError`, `WindowsError`, `socket.error`, `select.error` et `mmap.error` ont fusionnées en `OSError`, et le constructeur peut renvoyer une sous-classe.

Modifié dans la version 3.4 : L'attribut `filename` est maintenant le nom du fichier originel passé à la fonction, au lieu du nom encodé ou décodé à partir du *gestionnaire d'encodage et d'erreur du système de fichiers*. De plus, l'argument du constructeur et attribut *filename2* a été ajouté.

exception OverflowError

Raised when the result of an arithmetic operation is too large to be represented. This cannot occur for integers (which would rather raise `MemoryError` than give up). However, for historical reasons, OverflowError is sometimes raised for integers that are outside a required range. Because of the lack of standardization of floating-point exception handling in C, most floating-point operations are not checked.

exception PythonFinalizationError

This exception is derived from `RuntimeError`. It is raised when an operation is blocked during interpreter shutdown also known as *Python finalization*.

Examples of operations which can be blocked with a `PythonFinalizationError` during the Python finalization :

— Creating a new Python thread.

— `os.fork()`.
See also the `sys.is_finalizing()` function.

Ajouté dans la version 3.13 : Auparavant, une simple `RuntimeError` était levée.

exception RecursionError

Cette exception est dérivée de `RuntimeError`. Elle est levée lorsque l'interpréteur détecte que la profondeur de récursivité maximale (voir `sys.getrecursionlimit()`) est dépassée.

Ajouté dans la version 3.5 : Auparavant, une simple `RuntimeError` était levée.

exception ReferenceError

Cette exception est levée lorsqu'un pointeur faible d'un objet proxy, créé par la fonction `weakref.proxy()`, est utilisé pour accéder à un attribut du référent après qu'il ait été récupéré par le ramasse-miettes. Pour plus d'informations sur les pointeurs faibles, voir le module `weakref`.

exception RuntimeError

Levée lorsqu'une erreur qui n'appartient à aucune des autres catégories est détectée. La valeur associée est une chaîne de caractères indiquant précisément ce qui s'est mal passé.

exception StopIteration

Levée par la fonction native `next()` et la méthode `__next__()` d'un *iterator* (itérateur) pour signaler qu'il n'y a pas d'autres éléments produits par l'itérateur.

value

The exception object has a single attribute `value`, which is given as an argument when constructing the exception, and defaults to `None`.

Lorsqu'une fonction de type *generator* ou *coroutine* retourne une valeur, une nouvelle instance de `StopIteration` est levée, et la valeur retournée par la fonction est passée au paramètre `value` du constructeur de l'exception.

Si le code d'un générateur lève, directement ou indirectement, une `StopIteration`, elle est convertie en `RuntimeError` (en conservant `StopIteration` comme cause de la nouvelle exception).

Modifié dans la version 3.3 : Ajout de l'attribut `value` et de la possibilité pour les fonctions de générateur de l'utiliser pour retourner une valeur.

Modifié dans la version 3.5 : Introduit la transformation des erreurs `RuntimeError` via `from __future__ import generator_stop`, cf. **PEP 479**.

Modifié dans la version 3.7 : Active **PEP 479** pour tout le code par défaut : quand une erreur `StopIteration` est levée dans un générateur elle est transformée en une `RuntimeError`.

exception StopAsyncIteration

Must be raised by `__anext__()` method of an *asynchronous iterator* object to stop the iteration.

Ajouté dans la version 3.5.

exception SyntaxError(*message*, *details*)

Levée lorsque l'analyseur syntaxique rencontre une erreur de syntaxe. Cela peut se produire dans une instruction `import`, dans un appel aux fonctions natives `compile()`, `exec()` ou `eval()`, ou lors de la lecture du script initial ou de l'entrée standard (également de manière interactive).

La conversion en chaîne avec `str()` de l'instance de l'exception ne renvoie que le message d'erreur. L'argument `details` est un *n*-uplet dont les membres sont disponibles en tant qu'attributs séparés.

filename

Le nom du fichier dans lequel l'erreur de syntaxe a été rencontrée.

lineno

Le numéro de la ligne dans le fichier où l'erreur s'est produite. L'indiçage commence à 1 : `lineno` vaut 1 pour la première ligne du fichier.

offset

La colonne dans la ligne où l'erreur s'est produite. L'indiçage commence à 1 : `offset` vaut 1 pour le premier caractère de la ligne.

text

Le texte du code source impliqué dans l'erreur.

end_lineno

Le numéro de la dernière ligne produisant l'erreur. L'indiçage commence à 1 : `lineno` vaut 1 pour la première ligne du fichier.

end_offset

Le numéro de la dernière colonne (de la dernière ligne) produisant l'erreur. L'indiçage commence à 1 : `offset` vaut 1 pour le premier caractère de la ligne.

Pour les erreurs dans les chaînes de formatage *f-strings*, le message commence par « f-string : » et les champs *offset* sont les décalages dans un texte construit à partir de l'expression de remplacement. Par exemple, compiler `f'Bad {a b} field'` produit cet attribut *args* : `('f-string: ...', ('', 1, 2, '(a b)\n', 1, 5))`.

Modifié dans la version 3.10 : Ajout des attributs `end_lineno` et `end_offset`.

exception IndentationError

Classe mère pour les erreurs de syntaxe liées à une indentation incorrecte. C'est une sous-classe de `SyntaxError`.

exception TabError

Levée lorsqu'une indentation contient une utilisation incohérente des tabulations et des espaces. C'est une sous-classe de `IndentationError`.

exception SystemError

Raised when the interpreter finds an internal error, but the situation does not look so serious to cause it to abandon all hope. The associated value is a string indicating what went wrong (in low-level terms). In *CPython*, this could be raised by incorrectly using Python's C API, such as returning a `NULL` value without an exception set.

If you're confident that this exception wasn't your fault, or the fault of a package you're using, you should report this to the author or maintainer of your Python interpreter. Be sure to report the version of the Python interpreter (`sys.version`; it is also printed at the start of an interactive Python session), the exact error message (the exception's associated value) and if possible the source of the program that triggered the error.

exception SystemExit

Cette exception est levée par la fonction `sys.exit()`. Elle hérite de `BaseException` au lieu d'`Exception` pour ne pas qu'elle soit interceptée accidentellement par du code qui intercepte `Exception`. Cela permet à l'exception de se propager correctement et de faire quitter l'interpréteur. Lorsqu'elle n'est pas gérée, l'interpréteur Python quitte ; aucune trace d'appels n'est affichée. Le constructeur accepte le même argument optionnel passé à `sys.exit()`. Si la valeur est un entier, elle spécifie l'état de sortie du système (passé à la fonction C `exit()`) ; si elle est `None`, l'état de sortie est zéro ; si elle a un autre type (comme une chaîne de caractères), la valeur de l'objet est affichée et l'état de sortie est un.

Un appel à `sys.exit()` est traduit en une exception pour que les gestionnaires de nettoyage (les clauses `finally` des instructions `try`) puissent être exécutés, et pour qu'un débogueur puisse exécuter un script sans courir le risque de perdre le contrôle. La fonction `os._exit()` peut être utilisée s'il est absolument nécessaire de sortir immédiatement (par exemple, dans le processus enfant après un appel à `os.fork()`).

code

L'état de sortie ou le message d'erreur passé au constructeur. (`None` par défaut.)

exception TypeError

Levée lorsqu'une opération ou fonction est appliquée à un objet d'un type inapproprié. La valeur associée est une chaîne de caractères donnant des détails sur le type d'inadéquation.

Cette exception peut être levée par du code utilisateur pour indiquer qu'une tentative d'opération sur un objet n'est pas prise en charge, et n'est pas censée l'être. Si un objet est destiné à prendre en charge une opération donnée mais n'a pas encore fourni une implémentation, lever `NotImplementedError` est plus approprié.

Le passage d'arguments du mauvais type (e.g. passer une `list` quand un `int` est attendu) devrait résulter en un `TypeError`, mais le passage d'arguments avec la mauvaise valeur (e.g. un nombre en dehors des limites attendues) devrait résulter en une `ValueError`.

exception UnboundLocalError

Levée lorsqu'une référence est faite à une variable locale dans une fonction ou une méthode, mais qu'aucune valeur n'a été liée à cette variable. C'est une sous-classe de `NameError`.

exception UnicodeError

Levée lorsqu'une erreur d'encodage ou de décodage liée à Unicode se produit. C'est une sous-classe de `ValueError`.

`UnicodeError` a des attributs qui décrivent l'erreur d'encodage ou de décodage. Par exemple, `err.object[err.start:err.end]` donne l'entrée particulière invalide sur laquelle le codec a échoué.

encoding

Le nom de l'encodage qui a provoqué l'erreur.

reason

Une chaîne de caractères décrivant l'erreur de codec spécifique.

object

L'objet que le codec essayait d'encoder ou de décoder.

start

Le premier index des données invalides dans `object`.

This value should not be negative as it is interpreted as an absolute offset but this constraint is not enforced at runtime.

end

L'index après la dernière donnée invalide dans `object`.

This value should not be negative as it is interpreted as an absolute offset but this constraint is not enforced at runtime.

exception UnicodeEncodeError

Levée lorsqu'une erreur liée à Unicode se produit durant l'encodage. C'est une sous-classe d'`UnicodeError`.

exception UnicodeDecodeError

Levée lorsqu'une erreur liée à Unicode se produit durant le décodage. C'est une sous-classe d'`UnicodeError`.

exception UnicodeTranslateError

Levée lorsqu'une erreur liée à Unicode se produit durant la traduction. C'est une sous-classe d'`UnicodeError`.

exception ValueError

Levée lorsqu'une opération ou fonction native reçoit un argument qui possède le bon type mais une valeur inappropriée, et que la situation n'est pas décrite par une exception plus précise telle que `IndexError`.

exception ZeroDivisionError

Levée lorsque le second argument d'une opération de division ou d'un modulo est zéro. La valeur associée est une chaîne indiquant le type des opérandes et de l'opération.

Les exceptions suivantes sont conservées pour la compatibilité avec les anciennes versions ; depuis Python 3.3, ce sont des alias d'`OSError`.

exception EnvironmentError

exception IOError

exception WindowsError

Seulement disponible sous Windows.

5.4.1 Exceptions système

Les exceptions suivantes sont des sous-classes d'*OSError*, elles sont levées en fonction du code d'erreur système.

exception BlockingIOError

Raised when an operation would block on an object (e.g. socket) set for non-blocking operation. Corresponds to `errno` *EAGAIN*, *EALREADY*, *EWOULDBLOCK* and *EINPROGRESS*.

En plus de ceux de *OSError*, *BlockingIOError* peut avoir un attribut de plus :

characters_written

Un nombre entier contenant le nombre de caractères écrits dans le flux avant qu'il ne soit bloqué. Cet attribut est disponible lors de l'utilisation des classes tampon entrées-sorties du module *io*.

exception ChildProcessError

Raised when an operation on a child process failed. Corresponds to `errno` *ECHILD*.

exception ConnectionError

Classe mère pour les problèmes de connexion.

Les sous-classes sont *BrokenPipeError*, *ConnectionAbortedError*, *ConnectionRefusedError* et *ConnectionResetError*.

exception BrokenPipeError

A subclass of *ConnectionError*, raised when trying to write on a pipe while the other end has been closed, or trying to write on a socket which has been shutdown for writing. Corresponds to `errno` *EPIPE* and *ESHUTDOWN*.

exception ConnectionAbortedError

A subclass of *ConnectionError*, raised when a connection attempt is aborted by the peer. Corresponds to `errno` *ECONNABORTED*.

exception ConnectionRefusedError

A subclass of *ConnectionError*, raised when a connection attempt is refused by the peer. Corresponds to `errno` *ECONNREFUSED*.

exception ConnectionResetError

A subclass of *ConnectionError*, raised when a connection is reset by the peer. Corresponds to `errno` *ECONNRESET*.

exception FileExistsError

Raised when trying to create a file or directory which already exists. Corresponds to `errno` *EEXIST*.

exception FileNotFoundError

Raised when a file or directory is requested but doesn't exist. Corresponds to `errno` *ENOENT*.

exception InterruptedError

Raised when a system call is interrupted by an incoming signal. Corresponds to `errno` *EINTR*.

Modifié dans la version 3.5 : Python relance maintenant les appels système lorsqu'ils sont interrompus par un signal, sauf si le gestionnaire de signal lève une exception (voir **PEP 475** pour les raisons), au lieu de lever *InterruptedError*.

exception IsADirectoryError

Raised when a file operation (such as *os.remove()*) is requested on a directory. Corresponds to `errno` *EISDIR*.

exception NotADirectoryError

Raised when a directory operation (such as `os.listdir()`) is requested on something which is not a directory. On most POSIX platforms, it may also be raised if an operation attempts to open or traverse a non-directory file as if it were a directory. Corresponds to `errno` `ENOTDIR`.

exception PermissionError

Raised when trying to run an operation without the adequate access rights - for example filesystem permissions. Corresponds to `errno` `EACCES`, `EPERM`, and `ENOTCAPABLE`.

Modifié dans la version 3.11.1 : WASI's `ENOTCAPABLE` is now mapped to `PermissionError`.

exception ProcessLookupError

Raised when a given process doesn't exist. Corresponds to `errno` `ESRCH`.

exception TimeoutError

Raised when a system function timed out at the system level. Corresponds to `errno` `ETIMEDOUT`.

Ajouté dans la version 3.3 : Toutes les sous-classes d'`OSError` ci-dessus ont été ajoutées.

Voir aussi

PEP 3151 -- Refonte de la hiérarchie des exceptions système et IO

5.5 Avertissements

Les exceptions suivantes sont utilisées comme catégories d'avertissement ; voir `warning-categories` pour plus d'informations.

exception Warning

Classe mère pour les catégories d'avertissement.

exception UserWarning

Classe mère pour les avertissements générés par du code utilisateur.

exception DeprecationWarning

Classe mère pour les avertissements sur les fonctionnalités obsolètes, lorsque ces avertissements sont destinés aux autres développeurs Python.

Ignoré par les filtres d'avertissements par défaut, sauf dans le module `__main__` (**PEP 565**). Activer le *mode de développement de Python* affiche cet avertissement.

La politique relative à l'obsolescence est décrite dans la **PEP 387**.

exception PendingDeprecationWarning

Classe mère pour les avertissements d'obsolescence programmée. Ils indiquent que la fonctionnalité peut encore être utilisée actuellement, mais qu'elle sera supprimée dans le futur.

Cette classe est rarement utilisée car émettre un avertissement à propos d'une obsolescence à venir est inhabituel, et `DeprecationWarning` est préféré pour les obsolescences actuelles.

Ignoré par les filtres d'avertissements par défaut. Activer le *mode de développement de Python* affiche cet avertissement.

La politique relative à l'obsolescence est décrite dans la **PEP 387**.

exception SyntaxWarning

Classe mère pour les avertissements sur de la syntaxe douteuse.

exception RuntimeWarning

Classe mère pour les avertissements sur les comportements d'exécution douteux.

exception FutureWarning

Classe mère pour les avertissements à propos de fonctionnalités qui seront obsolètes dans le futur quand ces avertissements destinés aux utilisateurs finaux des applications écrites en Python.

exception ImportWarning

Classe mère pour les avertissements sur des erreurs probables dans les importations de modules.

Ignoré par les filtres d'avertissements par défaut. Activer le *mode de développement de Python* affiche cet avertissement.

exception UnicodeWarning

Classe mère pour les avertissements liés à l'Unicode.

exception EncodingWarning

Classe mère pour les avertissements liés à l'encodage des chaînes.

Voir *Opt-in EncodingWarning* pour plus d'informations.

Ajouté dans la version 3.10.

exception BytesWarning

Classe mère pour les avertissements liés à `bytes` et `bytearray`.

exception ResourceWarning

Classe mère pour les avertissements liés à l'utilisation des ressources.

Ignoré par les filtres d'avertissements par défaut. Activer le *mode de développement de Python* affiche cet avertissement.

Ajouté dans la version 3.2.

5.6 Exception groups

The following are used when it is necessary to raise multiple unrelated exceptions. They are part of the exception hierarchy so they can be handled with `except` like all other exceptions. In addition, they are recognised by `except*`, which matches their subgroups based on the types of the contained exceptions.

exception ExceptionGroup(*msg*, *excs*)

exception BaseExceptionGroup(*msg*, *excs*)

Both of these exception types wrap the exceptions in the sequence `excs`. The `msg` parameter must be a string. The difference between the two classes is that `BaseExceptionGroup` extends `BaseException` and it can wrap any exception, while `ExceptionGroup` extends `Exception` and it can only wrap subclasses of `Exception`. This design is so that `except Exception` catches an `ExceptionGroup` but not `BaseExceptionGroup`.

The `BaseExceptionGroup` constructor returns an `ExceptionGroup` rather than a `BaseExceptionGroup` if all contained exceptions are `Exception` instances, so it can be used to make the selection automatic. The `ExceptionGroup` constructor, on the other hand, raises a `TypeError` if any contained exception is not an `Exception` subclass.

message

The `msg` argument to the constructor. This is a read-only attribute.

exceptions

A tuple of the exceptions in the `excs` sequence given to the constructor. This is a read-only attribute.

subgroup(*condition*)

Returns an exception group that contains only the exceptions from the current group that match *condition*, or `None` if the result is empty.

The condition can be an exception type or tuple of exception types, in which case each exception is checked for a match using the same check that is used in an `except` clause. The condition can also be

a callable (other than a type object) that accepts an exception as its single argument and returns true for the exceptions that should be in the subgroup.

The nesting structure of the current exception is preserved in the result, as are the values of its *message*, *__traceback__*, *__cause__*, *__context__* and *__notes__* fields. Empty nested groups are omitted from the result.

The condition is checked for all exceptions in the nested exception group, including the top-level and any nested exception groups. If the condition is true for such an exception group, it is included in the result in full.

Ajouté dans la version 3.13 : `condition` can be any callable which is not a type object.

split(*condition*)

Like *subgroup()*, but returns the pair `(match, rest)` where `match` is `subgroup(condition)` and `rest` is the remaining non-matching part.

derive(*excs*)

Returns an exception group with the same *message*, but which wraps the exceptions in `excs`.

This method is used by *subgroup()* and *split()*, which are used in various contexts to break up an exception group. A subclass needs to override it in order to make *subgroup()* and *split()* return instances of the subclass rather than *ExceptionGroup*.

subgroup() and *split()* copy the *__traceback__*, *__cause__*, *__context__* and *__notes__* fields from the original exception group to the one returned by *derive()*, so these fields do not need to be updated by *derive()*.

```
>>> class MyGroup(ExceptionGroup):
...     def derive(self, excs):
...         return MyGroup(self.message, excs)
...
>>> e = MyGroup("eg", [ValueError(1), TypeError(2)])
>>> e.add_note("a note")
>>> e.__context__ = Exception("context")
>>> e.__cause__ = Exception("cause")
>>> try:
...    raise e
... except Exception as e:
...    exc = e
...
>>> match, rest = exc.split(ValueError)
>>> exc, exc.__context__, exc.__cause__, exc.__notes__
(MyGroup('eg', [ValueError(1), TypeError(2)]), Exception('context'),␣
↪Exception('cause'), ['a note'])
>>> match, match.__context__, match.__cause__, match.__notes__
(MyGroup('eg', [ValueError(1)]), Exception('context'), Exception('cause'),␣
↪['a note'])
>>> rest, rest.__context__, rest.__cause__, rest.__notes__
(MyGroup('eg', [TypeError(2)]), Exception('context'), Exception('cause'), [
↪'a note'])
>>> exc.__traceback__ is match.__traceback__ is rest.__traceback__
True
```

Note that *BaseExceptionGroup* defines `__new__()`, so subclasses that need a different constructor signature need to override that rather than `__init__()`. For example, the following defines an exception group subclass which accepts an exit_code and and constructs the group's message from it.

```
class Errors(ExceptionGroup):
   def __new__(cls, errors, exit_code):
      self = super().__new__(Errors, f"exit code: {exit_code}", errors)
```

(suite sur la page suivante)

(suite de la page précédente)

```
        self.exit_code = exit_code
        return self

    def derive(self, excs):
        return Errors(excs, self.exit_code)
```

Like *ExceptionGroup*, any subclass of *BaseExceptionGroup* which is also a subclass of *Exception* can only wrap instances of *Exception*.

Ajouté dans la version 3.11.

5.7 Hiérarchie des exceptions

La hiérarchie de classes pour les exceptions natives est la suivante :

```
BaseException
 ├── BaseExceptionGroup
 ├── GeneratorExit
 ├── KeyboardInterrupt
 ├── SystemExit
 └── Exception
      ├── ArithmeticError
      │    ├── FloatingPointError
      │    ├── OverflowError
      │    └── ZeroDivisionError
      ├── AssertionError
      ├── AttributeError
      ├── BufferError
      ├── EOFError
      ├── ExceptionGroup [BaseExceptionGroup]
      ├── ImportError
      │    └── ModuleNotFoundError
      ├── LookupError
      │    ├── IndexError
      │    └── KeyError
      ├── MemoryError
      ├── NameError
      │    └── UnboundLocalError
      ├── OSError
      │    ├── BlockingIOError
      │    ├── ChildProcessError
      │    ├── ConnectionError
      │    │    ├── BrokenPipeError
      │    │    ├── ConnectionAbortedError
      │    │    ├── ConnectionRefusedError
      │    │    └── ConnectionResetError
      │    ├── FileExistsError
      │    ├── FileNotFoundError
      │    ├── InterruptedError
      │    ├── IsADirectoryError
      │    ├── NotADirectoryError
      │    ├── PermissionError
      │    ├── ProcessLookupError
      │    └── TimeoutError
      ├── ReferenceError
      ├── RuntimeError
```

(suite sur la page suivante)

(suite de la page précédente)

```
      │     ├── NotImplementedError
      │     ├── PythonFinalizationError
      │     └── RecursionError
      ├── StopAsyncIteration
      ├── StopIteration
      ├── SyntaxError
      │     └── IndentationError
      │          └── TabError
      ├── SystemError
      ├── TypeError
      ├── ValueError
      │     └── UnicodeError
      │          ├── UnicodeDecodeError
      │          ├── UnicodeEncodeError
      │          └── UnicodeTranslateError
      └── Warning
            ├── BytesWarning
            ├── DeprecationWarning
            ├── EncodingWarning
            ├── FutureWarning
            ├── ImportWarning
            ├── PendingDeprecationWarning
            ├── ResourceWarning
            ├── RuntimeWarning
            ├── SyntaxWarning
            ├── UnicodeWarning
            └── UserWarning
```

CHAPITRE 6

Services de Manipulation de Texte

Les modules décrits dans ce chapitre fournissent un large ensemble d'opérations de manipulation sur les chaînes de caractères et le texte en général.

Le module `codecs` documenté dans *Services autour des Données Binaires* est aussi très pertinent pour la manipulation de texte. Consultez aussi la documentation du type *str* natif Python dans *Type Séquence de Texte — str*.

6.1 `string` --- Common string operations

Code source : Lib/string.py

Voir aussi

Type Séquence de Texte — str

Méthodes de chaînes de caractères

6.1.1 Chaînes constantes

Les constantes définies dans ce module sont :

`string.`**`ascii_letters`**

La concaténation des constantes `ascii_lowercase` et `ascii_uppercase` décrites ci-dessous. Cette valeur n'est pas dépendante de la configuration de la localisation.

`string.`**`ascii_lowercase`**

Les lettres minuscules `'abcdefghijklmnopqrstuvwxyz'`. Cette valeur ne dépend pas de la configuration de localisation et ne changera pas.

`string.`**`ascii_uppercase`**

Les lettres majuscules `'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`. Cette valeur ne dépend pas de la configuration de la localisation et ne changera pas.

`string.`**`digits`**

La chaîne `'0123456789'`.

string.**hexdigits**

La chaîne `'0123456789abcdefABCDEF'`.

string.**octdigits**

La chaîne `'01234567`.

string.**punctuation**

Chaîne de caractères ASCII considérés comme ponctuation dans la configuration de localisation `C` : `! "#$%&'()*+,-./:;<=>?@[\]^_`{|}~`.

string.**printable**

String of ASCII characters which are considered printable by Python. This is a combination of *digits*, *ascii_letters*, *punctuation*, and *whitespace*.

> **Note**
>
> By design, *string.printable.isprintable()* returns *False*. In particular, `string.printable` is not printable in the POSIX sense (see *LC_CTYPE*).

string.**whitespace**

Une chaîne comprenant tous les caractères ASCII considérés comme espaces. Sont inclus les caractères espace, tabulations, saut de ligne, retour du chariot, saut de page, et tabulation verticale.

6.1.2 Formatage personnalisé de chaîne

La classe primitive `str` fournit la possibilité de faire des substitutions de variables complexes et du formatage de valeurs via la méthode *format()* décrite par **PEP 3101**. La classe *Formatter* dans le module *string* permet de créer et personnaliser vos propres comportements de formatage de chaînes en utilisant la même implémentation que la méthode primitive *format()*.

class string.**Formatter**

La classe *Formatter* a les méthodes publiques suivantes :

format(*format_string*, /, **args*, ***kwargs*)

La méthode principale de l'API. Elle prend une chaîne de format et un ensemble arbitraire d'arguments positions et mot-clefs. C'est uniquement un conteneur qui appelle *vformat()*.

Modifié dans la version 3.7 : L'argument *format_string* est maintenant *obligatoirement un paramètre positionnel*.

vformat(*format_string*, *args*, *kwargs*)

Cette fonction fait le travail effectif du formatage. Elle existe comme méthode séparée au cas où vous voudriez passer un dictionnaire d'arguments prédéfini plutôt que décompresser et recompresser le dictionnaire en arguments individuels en utilisant la syntaxe `*args` et `**kwargs`. *vformat()* s'occupe de découper la chaîne de format en données de caractères et champs de remplacement. Elle appelle les différentes méthodes décrites ci-dessous.

De plus, la classe *Formatter* définit un certain nombre de méthodes qui ont pour vocation d'être remplacées par des sous-classes :

parse(*format_string*)

Boucle sur la chaîne de format et renvoie un itérable de quadruplets (*literal_text*, *field_name*, *format_spec*, *conversion*). Ceci est utilisé par *vformat()* pour découper la chaîne de format en littéraux ou en champs de remplacement.

Les valeurs dans le *n*-uplet représentent conceptuellement un ensemble de littéraux suivis d'un unique champ de remplacement. S'il n'y a pas de littéral, (ce qui peut arriver si deux champs de remplacement sont placés côte à côte), alors *literal_text* est une chaîne vide. S'il n'y a pas de champ de remplacement, les valeurs *field_name*, *format_spec* et *conversion* sont mises à `None`.

get_field(*field_name*, *args*, *kwargs*)

Récupère le champ *field_name* du *n*-uplet renvoyé par `parse()` (voir ci-dessus), le convertit en un objet à formater. Renvoie une paire (*obj*, *used_key*). La version par défaut prend une chaîne de la forme définie par **PEP 3101**, telle que `"0[name]"` ou `"label.title"`. *args* et *kwargs* sont tels que ceux passés à `vformat()`. La valeur renvoyée *used_key* a le même sens que le paramètre *key* de `get_value()`.

get_value(*key*, *args*, *kwargs*)

Récupère la valeur d'un champ donné. L'argument *key* est soit un entier, soit une chaîne. Si c'est un entier, il représente l'indice de l'argument dans *args*. Si c'est une chaîne de caractères, elle représente le nom de l'argument dans *kwargs*.

Le paramètre *args* est défini par la liste des arguments positionnels de `vformat()`, et le paramètre *kwargs* est défini par le dictionnaire des arguments mot-clefs.

Pour les noms de champs composés, ces fonctions sont uniquement appelées sur la première composante du nom. Les composantes suivantes sont manipulées au travers des attributs normaux et des opérations sur les indices.

Donc par exemple, le champ-expression `0.name` amènerait `get_value()` à être appelée avec un argument *key* valant 0. L'attribut `name` sera recherché après l'appel `get_value()` en faisant appel à la primitive `getattr()`.

Si l'indice ou le mot-clef fait référence à un objet qui n'existe pas, alors une exception `IndexError` ou `KeyError` doit être levée.

check_unused_args(*used_args*, *args*, *kwargs*)

Implémente une vérification pour les arguments non utilisés si désiré. L'argument de cette fonction est l'ensemble des clefs qui ont été effectivement référencées dans la chaîne de format (des entiers pour les indices et des chaînes de caractères pour les arguments nommés), et une référence vers les arguments *args* et *kwargs* qui ont été passés à `vformat`. L'ensemble des arguments non utilisés peut être calculé sur base de ces paramètres. `check_unused_args()` est censée lever une exception si la vérification échoue.

format_field(*value*, *format_spec*)

La méthode `format_field()` fait simplement appel à la primitive globale `format()`. Cette méthode est fournie afin que les sous-classes puissent la redéfinir.

convert_field(*value*, *conversion*)

Convertit la valeur (renvoyée par `get_field()`) selon un type de conversion donné (comme dans le *n*-uplet renvoyé par la méthode `parse()`). La version par défaut comprend 's' (`str`), 'r' (`repr`) et 'a' (ASCII) comme types de conversion.

6.1.3 Syntaxe de formatage de chaîne

La méthode `str.format()` et la classe `Formatter` partagent la même syntaxe pour les chaînes de formatage (bien que dans le cas de `Formatter` les sous-classes puissent définir leur propre syntaxe). La syntaxe est liée à celle des chaînes de formatage littérales, mais elle est moins sophistiquée, et surtout ne permet pas d'inclure des expressions arbitraires.

Les chaînes de formatage contiennent des "champs de remplacement" entourés d'accolades `{}`. Tout ce qui n'est pas placé entre deux accolades est considéré comme littéral, qui est copié tel quel dans le résultat. Si vous avez besoin d'inclure une accolade en littéral, elles peuvent être échappées en les doublant : `{{` et `}}`.

La grammaire pour un champ de remplacement est définie comme suit :

```
replacement_field: "{" [field_name] ["!" conversion] [":" format_spec] "}"
field_name:        arg_name ("." attribute_name | "[" element_index "]")*
arg_name:          [identifier | digit+]
attribute_name:    identifier
element_index:     digit+ | index_string
index_string:      <any source character except "]"> +
```

conversion: "r" | "s" | "a"
format_spec: *format-spec:format_spec*

En termes moins formels, un champ de remplacement peut débuter par un *field_name* qui spécifie l'objet dont la valeur va être formatée et insérée dans le résultat à l'endroit du champ de remplacement Le *field_name* peut éventuellement être suivi d'un champ *conversion* qui est précédé d'un point d'exclamation `'!'`, et d'un *format_spec* qui est précédé par deux-points `':'`. Cela définit un format personnalisé pour le remplacement d'une valeur.

Voir également la section *Mini-langage de spécification de format*.

The *field_name* itself begins with an *arg_name* that is either a number or a keyword. If it's a number, it refers to a positional argument, and if it's a keyword, it refers to a named keyword argument. An *arg_name* is treated as a number if a call to `str.isdecimal()` on the string would return true. If the numerical arg_names in a format string are 0, 1, 2, ... in sequence, they can all be omitted (not just some) and the numbers 0, 1, 2, ... will be automatically inserted in that order. Because *arg_name* is not quote-delimited, it is not possible to specify arbitrary dictionary keys (e.g., the strings `'10'` or `':-]'`) within a format string. The *arg_name* can be followed by any number of index or attribute expressions. An expression of the form `'.name'` selects the named attribute using `getattr()`, while an expression of the form `'[index]'` does an index lookup using `__getitem__()`.

Modifié dans la version 3.1 : Les spécificateurs de position d'argument peuvent être omis dans les appels à `str.format()`. Donc `'{} {}'.format(a, b)` est équivalent à `'{0} {1}'.format(a, b)`.

Modifié dans la version 3.4 : Les spécificateurs de position d'argument peuvent être omis pour `Formatter`.

Quelques exemples simples de formatage de chaînes :

```
"First, thou shalt count to {0}"  # References first positional argument
"Bring me a {}"                   # Implicitly references the first positional␣
↪argument
"From {} to {}"                   # Same as "From {0} to {1}"
"My quest is {name}"              # References keyword argument 'name'
"Weight in tons {0.weight}"       # 'weight' attribute of first positional arg
"Units destroyed: {players[0]}"   # First element of keyword argument 'players'.
```

The *conversion* field causes a type coercion before formatting. Normally, the job of formatting a value is done by the `__format__()` method of the value itself. However, in some cases it is desirable to force a type to be formatted as a string, overriding its own definition of formatting. By converting the value to a string before calling `__format__()`, the normal formatting logic is bypassed.

Actuellement, trois indicateurs sont gérés : `'!s'` qui appelle la fonction `str()` sur la valeur, `'!r'` qui appelle la fonction `repr()` et `!a` qui appelle la fonction `ascii()`.

Quelques exemples :

```
"Harold's a clever {0!s}"        # Calls str() on the argument first
"Bring out the holy {name!r}"    # Calls repr() on the argument first
"More {!a}"                      # Calls ascii() on the argument first
```

Le champ *format_spec* contient une spécification sur la manière selon laquelle la valeur devrait être représentée : des informations telles que la longueur du champ, l'alignement, le remplissage, la précision décimale, etc. Chaque type peut définir son propre "mini-langage de formatage" ou sa propre interprétation de *format_spec*.

La plupart des types natifs gèrent un mini-langage de formatage usuel qui est décrit dans la section suivante.

Un champ *format_spec* peut contenir un champ de remplacement imbriqué. Ces champs de remplacement imbriqués peuvent contenir un nom de champ, un indicateur de conversion, mais une imbrication récursive plus profonde n'est pas permise. Les champs de remplacement au sein de *format_spec* sont substitués avant que la chaîne *format_spec* ne soit interprétée. Cela permet que le formatage d'une valeur soit dynamiquement spécifié.

Voir la section *Exemples de formats* pour des exemples.

Mini-langage de spécification de format

Les "Spécifications de format" sont utilisées avec les champs de remplacement contenus dans les chaînes de formatage, pour définir comment les valeurs doivent être représentées (voir *Syntaxe de formatage de chaîne* et f-strings). Elles peuvent aussi être passées directement à la fonction native `format()`. Chaque type *formatable* peut définir comment les spécifications sur les valeurs de ce type doivent être interprétées.

La plupart des primitives implémentent les options suivantes, même si certaines options de formatage ne sont prises en charge que pour les types numériques.

Une convention généralement admise est qu'une chaîne vide produit le même résultat que si vous aviez appelé `str()` sur la valeur. Une chaîne de format non vide modifie habituellement le résultat.

La forme générale d'un *spécificateur de format standard* est :

```
format_spec:     [[fill]align][sign]["z"]["#"]["0"][width][grouping_option]["."␣
↪precision][type]
fill:            <any character>
align:           "<" | ">" | "=" | "^"
sign:            "+" | "-" | " "
width:           digit+
grouping_option: "_" | ","
precision:       digit+
type:            "b" | "c" | "d" | "e" | "E" | "f" | "F" | "g" | "G" | "n" | "o" |␣
↪"s" | "x" | "X" | "%"
```

Si une valeur valide est spécifiée pour *align*, elle peut être précédée par un caractère *fill*, qui peut être n'importe quel caractère (espace par défaut si la valeur est omise). Il n'est pas possible d'utiliser une accolade littérale ("`{`" ou "`}`") comme caractère *fill* dans une chaîne de formatage littérale ou avec la méthode `str.format()`. Cependant, il est possible d'insérer une accolade à l'aide d'un champ de remplacement imbriqué. Cette limitation n'affecte pas la fonction `format()`.

Le sens des différentes options d'alignement est donné comme suit :

Option	Signification
`'<'`	Force le champ à être aligné à gauche dans l'espace disponible (c'est le spécificateur par défaut pour la plupart des objets).
`'>'`	Force le champ à être aligné à droite dans l'espace disponible (c'est le spécificateur par défaut pour les nombres).
`'='`	Forces the padding to be placed after the sign (if any) but before the digits. This is used for printing fields in the form '+000000120'. This alignment option is only valid for numeric types, excluding `complex`. It becomes the default for numbers when '0' immediately precedes the field width.
`'^'`	Force le champ à être centré dans l'espace disponible.

Notons que la longueur du champ est toujours égale à la taille nécessaire pour remplir le champ avec l'objet à moins que la valeur minimum ne soit précisée. Ainsi, si aucune valeur n'est précisée, l'option d'alignement n'a aucun sens.

L'option *sign* est uniquement valide pour les types numériques, et peut valoir :

Option	Signification
`'+'`	indique que le signe doit être affiché pour les nombres tant positifs que négatifs.
`'-'`	indique que le signe doit être affiché uniquement pour les nombres négatifs (c'est le comportement par défaut).
espace	indique qu'un espace doit précéder les nombres positifs et qu'un signe moins doit précéder les nombres négatifs.

The `'z'` option coerces negative zero floating-point values to positive zero after rounding to the format precision. This option is only valid for floating-point presentation types.

Modifié dans la version 3.11 : Added the `'z'` option (see also **PEP 682**).

L'option `'#'` impose l'utilisation de la "forme alternative" pour la conversion. La forme alternative est définie différemment pour différent types. Cette option est uniquement valide pour les types entiers, flottants et complexes. Pour les entiers, quand l'affichage binaire, octal ou hexadécimal est utilisé, cette option ajoute le préfixe `'0b'`, `'0o'`, `'0x'` ou `'0X'` à la valeur affichée. Pour les flottants et les complexes, la forme alternative impose que le résultat de la conversion contienne toujours une virgule, même si aucun chiffre ne vient après. Normalement, une virgule apparaît dans le résultat de ces conversions seulement si un chiffre le suit. De plus, pour les conversions `'g'` et `'G'`, les zéros finaux ne sont pas retirés du résultat.

The `','` option signals the use of a comma for a thousands separator for floating-point presentation types and for integer presentation type `'d'`. For other presentation types, this option is an error. For a locale aware separator, use the `'n'` integer presentation type instead.

Modifié dans la version 3.1 : Ajout de l'option `','` (voir **PEP 378**).

The `'_'` option signals the use of an underscore for a thousands separator for floating-point presentation types and for integer presentation type `'d'`. For integer presentation types `'b'`, `'o'`, `'x'`, and `'X'`, underscores will be inserted every 4 digits. For other presentation types, specifying this option is an error.

Modifié dans la version 3.6 : Ajout de l'option `'_'` (voir aussi **PEP 515**).

width est un entier en base 10 qui définit la longueur minimale de tout le champ, y compris les préfixes, séparateurs et autres caractères de formatage. Si elle n'est pas spécifiée, alors le champ *width* est déterminé par le contenu.

When no explicit alignment is given, preceding the *width* field by a zero (`'0'`) character enables sign-aware zero-padding for numeric types, excluding `complex`. This is equivalent to a *fill* character of `'0'` with an *alignment* type of `'='`.

Modifié dans la version 3.10 : Preceding the *width* field by `'0'` no longer affects the default alignment for strings.

The *precision* is a decimal integer indicating how many digits should be displayed after the decimal point for presentation types `'f'` and `'F'`, or before and after the decimal point for presentation types `'g'` or `'G'`. For string presentation types the field indicates the maximum field size - in other words, how many characters will be used from the field content. The *precision* is not allowed for integer presentation types.

Finalement, le spécificateur *type* détermine comment la donnée doit être représentée.

Les types disponibles de représentation de chaîne sont :

Type	Signification
`'s'`	Format de chaîne. C'est le type par défaut pour les chaînes de caractères et peut être omis.
`None`	Pareil que `'s'`.

Les types disponibles de représentation d'entier sont :

Type	Signification
`'b'`	Format binaire. Affiche le nombre en base 2.
`'c'`	Caractère. Convertit l'entier en caractère Unicode associé avant de l'afficher.
`'d'`	Entier décimal. Affiche le nombre en base 10.
`'o'`	Format octal. Affiche le nombre en base 8.
`'x'`	Format hexadécimal. Affiche le nombre en base 16 en utilisant les lettres minuscules pour les chiffres au-dessus de 9.
`'X'`	Format hexadécimal. Affiche le nombre en base 16 en utilisant les lettres majuscules pour les chiffres au-dessus de 9. Si `'#'` est présent, le préfixe `'0x'` est également passé en majuscules pour devenir `'0X'`.
`'n'`	Nombre. Pareil que `'d'` si ce n'est que ce sont les paramètres de localisation qui sont utilisés afin de déterminer le séparateur de nombres approprié.
`None`	Pareil que `'d'`.

In addition to the above presentation types, integers can be formatted with the floating-point presentation types listed below (except `'n'` and `None`). When doing so, `float()` is used to convert the integer to a floating-point number before formatting.

Les représentations possibles pour les `float` et les `Decimal` sont :

Type	Signification
`'e'`	Scientific notation. For a given precision `p`, formats the number in scientific notation with the letter 'e' separating the coefficient from the exponent. The coefficient has one digit before and `p` digits after the decimal point, for a total of `p + 1` significant digits. With no precision given, uses a precision of `6` digits after the decimal point for `float`, and shows all coefficient digits for `Decimal`. If `p=0`, the decimal point is omitted unless the `#` option is used.
`'E'`	Notation scientifique. Pareil que `'e'` sauf que Python utilise la lettre majuscule 'E' comme séparateur.
`'f'`	Fixed-point notation. For a given precision `p`, formats the number as a decimal number with exactly `p` digits following the decimal point. With no precision given, uses a precision of `6` digits after the decimal point for `float`, and uses a precision large enough to show all coefficient digits for `Decimal`. If `p=0`, the decimal point is omitted unless the `#` option is used.
`'F'`	Virgule fixe. Pareil que `'f'` à part `nan` qui devient `NAN` et `inf` qui devient `INF`.
`'g'`	Format général. Pour une précision donnée `p >= 1`, Python arrondit le nombre à `p` chiffres significatifs et puis formate le résultat soit en virgule fixe soit en notation scientifique, en fonction de la magnitude. Une précision de `0` est considérée équivalente à une précision de `1`. Les règles précises sont les suivantes : supposons que le résultat formaté avec le type de représentation `'e'` et une précision `p-1` ait un exposant `exp`. Alors, si `m <= exp <= p` où `m` vaut `-4` pour les nombres à virgule flottante et `-6` pour les `Decimals`, le nombre est formaté avec le type de représentation `'f'` et une précision `p-1-exp`. Sinon, le nombre est formaté avec le type de représentation `'e'` et une précision `p-1`. Dans les deux cas, les zéros finaux et non significatifs sont retirés, et la virgule est également retirée s'il n'y a aucun chiffre la suivant, sauf si l'option `'#'` est utilisée. Si la précision n'est pas donnée, utilise une précision de `6` chiffres significatifs pour les `float`. Pour les `Decimal`, la mantisse du résultat dépend de la valeur : la notation scientifique est utilisée pour les valeurs inférieures à `1e-6` en valeur absolue et pour celles dont le dernier chiffre significatif a une position supérieure à 1 ; dans les autres cas, la notation en virgule fixe est utilisée. Les valeurs suivantes : infini négatif, infini positif, zéro positif, zéro négatif, *not a number* sont formatées respectivement par `inf`, `-inf`, `0`, `-0` et `nan`, peu importe la précision.
`'G'`	Format général. Pareil que `'G'` si ce n'est que `'E'` est utilisé si le nombre est trop grand. La représentation des infinis et de *NaN* sont en majuscules également.
`'n'`	Nombre. Pareil que `'g'`, si ce n'est que la configuration de localisation est prise en compte pour insérer le séparateur approprié.
`'%'`	Pourcentage. Multiplie le nombre par 100 et l'affiche en virgule fixe (`'f'`), suivi d'un symbole pourcent `'%'`.
`None`	For `float` this is like the `'g'` type, except that when fixed-point notation is used to format the result, it always includes at least one digit past the decimal point, and switches to the scientific notation when `exp >= p - 1`. When the precision is not specified, the latter will be as large as needed to represent the given value faithfully. Pour les `Decimal`, c'est identique à `'g'` ou `'G'` en fonction de la valeur de `context.capitals` du contexte décimal courant. L'effet visé est de coller à la valeur renvoyée par `str()` telle que modifiée par les autres modificateurs de format.

The result should be correctly rounded to a given precision `p` of digits after the decimal point. The rounding mode for `float` matches that of the `round()` builtin. For `Decimal`, the rounding mode of the current *context* will be used.

The available presentation types for `complex` are the same as those for `float` (`'%'` is not allowed). Both the real and imaginary components of a complex number are formatted as floating-point numbers, according to the specified presentation type. They are separated by the mandatory sign of the imaginary part, the latter being terminated by a `j` suffix. If the presentation type is missing, the result will match the output of `str()` (complex numbers with a

non-zero real part are also surrounded by parentheses), possibly altered by other format modifiers.

Exemples de formats

Cette section contient des exemples de la syntaxe de `str.format()` et des comparaisons avec l'ancien formatage par `%`.

Dans la plupart des cas, la syntaxe est similaire à l'ancien formatage par `%`, avec l'ajout de `{}` et avec `:` au lieu de `%`. Par exemple : `'%03.2f'` peut être changé en `'{03.2f}'`.

La nouvelle syntaxe de formatage gère également de nouvelles options et des options différentes, montrées dans les exemples qui suivent.

Accéder à un argument par sa position :

```
>>> '{0}, {1}, {2}'.format('a', 'b', 'c')
'a, b, c'
>>> '{}, {}, {}'.format('a', 'b', 'c')  # 3.1+ only
'a, b, c'
>>> '{2}, {1}, {0}'.format('a', 'b', 'c')
'c, b, a'
>>> '{2}, {1}, {0}'.format(*'abc')      # unpacking argument sequence
'c, b, a'
>>> '{0}{1}{0}'.format('abra', 'cad')   # arguments' indices can be repeated
'abracadabra'
```

Accéder à un argument par son nom :

```
>>> 'Coordinates: {latitude}, {longitude}'.format(latitude='37.24N', longitude='-
↪115.81W')
'Coordinates: 37.24N, -115.81W'
>>> coord = {'latitude': '37.24N', 'longitude': '-115.81W'}
>>> 'Coordinates: {latitude}, {longitude}'.format(**coord)
'Coordinates: 37.24N, -115.81W'
```

Accéder aux attributs d'un argument :

```
>>> c = 3-5j
>>> ('The complex number {0} is formed from the real part {0.real} '
...  'and the imaginary part {0.imag}.').format(c)
'The complex number (3-5j) is formed from the real part 3.0 and the imaginary part␣
↪-5.0.'
>>> class Point:
...     def __init__(self, x, y):
...         self.x, self.y = x, y
...     def __str__(self):
...         return 'Point({self.x}, {self.y})'.format(self=self)
...
>>> str(Point(4, 2))
'Point(4, 2)'
```

Accéder aux éléments d'un argument :

```
>>> coord = (3, 5)
>>> 'X: {0[0]};  Y: {0[1]}'.format(coord)
'X: 3;  Y: 5'
```

Remplacer `%s` et `%r` :

```
>>> "repr() shows quotes: {!r}; str() doesn't: {!s}".format('test1', 'test2')
"repr() shows quotes: 'test1'; str() doesn't: test2"
```

Aligner le texte et spécifier une longueur minimale :

```
>>> '{:<30}'.format('left aligned')
'left aligned                  '
>>> '{:>30}'.format('right aligned')
'                 right aligned'
>>> '{:^30}'.format('centered')
'           centered           '
>>> '{:*^30}'.format('centered')  # use '*' as a fill char
'***********centered***********'
```

Remplacer `%+f`, `%-f`, et `%f` et spécifier un signe :

```
>>> '{:+f}; {:+f}'.format(3.14, -3.14)  # show it always
'+3.140000; -3.140000'
>>> '{: f}; {: f}'.format(3.14, -3.14)  # show a space for positive numbers
' 3.140000; -3.140000'
>>> '{:-f}; {:-f}'.format(3.14, -3.14)  # show only the minus -- same as '{:f};
↪{:f}'
'3.140000; -3.140000'
```

Remplacer `%x` et `%o` et convertir la valeur dans différentes bases :

```
>>> # format also supports binary numbers
>>> "int: {0:d};  hex: {0:x};  oct: {0:o};  bin: {0:b}".format(42)
'int: 42;  hex: 2a;  oct: 52;  bin: 101010'
>>> # with 0x, 0o, or 0b as prefix:
>>> "int: {0:d};  hex: {0:#x};  oct: {0:#o};  bin: {0:#b}".format(42)
'int: 42;  hex: 0x2a;  oct: 0o52;  bin: 0b101010'
```

Utiliser une virgule comme séparateur des milliers :

```
>>> '{:,}'.format(1234567890)
'1,234,567,890'
```

Exprimer un pourcentage :

```
>>> points = 19
>>> total = 22
>>> 'Correct answers: {:.2%}'.format(points/total)
'Correct answers: 86.36%'
```

Utiliser un formatage propre au type :

```
>>> import datetime
>>> d = datetime.datetime(2010, 7, 4, 12, 15, 58)
>>> '{:%Y-%m-%d %H:%M:%S}'.format(d)
'2010-07-04 12:15:58'
```

Arguments imbriqués et des exemples plus complexes :

```
>>> for align, text in zip('<^>', ['left', 'center', 'right']):
...     '{0:{fill}{align}16}'.format(text, fill=align, align=align)
...
'left<<<<<<<<<<<<'
'^^^^^center^^^^^'
'>>>>>>>>>>>right'
>>>
>>> octets = [192, 168, 0, 1]
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> '{:02X}{:02X}{:02X}{:02X}'.format(*octets)
'C0A80001'
>>> int(_, 16)
3232235521
>>>
>>> width = 5
>>> for num in range(5,12):
...     for base in 'dXob':
...         print('{0:{width}{base}}'.format(num, base=base, width=width), end=' ')
...     print()
...
    5     5     5   101
    6     6     6   110
    7     7     7   111
    8     8    10  1000
    9     9    11  1001
   10     A    12  1010
   11     B    13  1011
```

6.1.4 Chaînes modèles

Template strings provide simpler string substitutions as described in **PEP 292**. A primary use case for template strings is for internationalization (i18n) since in that context, the simpler syntax and functionality makes it easier to translate than other built-in string formatting facilities in Python. As an example of a library built on template strings for i18n, see the flufl.i18n package.

Les chaînes modèles prennent en charge les substitutions basées sur `$` en utilisant les règles suivantes :

- `$$` est un échappement ; il est remplacé par un simple `$`.
- `$identifier` dénomme un substituant lié à la clef `"identifier"`. Par défaut, `"identifier"` est restreint à toute chaîne de caractères ASCII alphanumériques sensibles à la casse (avec les *underscores*) commençant avec un *underscore* ou un caractère alphanumérique. Le premier caractère n'étant pas un identifieur après le `$` termine la spécification du substituant.
- `${identifier}` est équivalent à `$identifier`. Cette notation est requise quand un caractère valide pour une clef de substituant suit directement le substituant mais ne fait pas partie du substituant, comme `"${noun}ification"`.

Tout autre présence du symbole `$` dans une chaîne résultera en la levée d'une `ValueError`.

Le module `string` fournit une classe `Template` qui implémente ces règles. Les méthodes de `Template` sont :

class `string.`**Template**(*template*)

Le constructeur prend un seul argument qui est la chaîne du *template*.

substitute(*mapping={}*, */*, ***kwds*)

Applique les substitutions du *template*, et la renvoie dans une nouvelle chaîne. *mapping* est un objet dictionnaire-compatible qui lie les substituants dans le *template*. De même, vous pouvez fournir des arguments mot-clefs tels que les mot-clefs sont les substituants. Quand à la fois *mapping* et *kwds* sont donnés et qu'il y a des doublons, les substituants de *kwds* sont prioritaires.

safe_substitute(*mapping={}*, */*, ***kwds*)

Comme `substitute()`, si ce n'est qu'au lieu de lever une `KeyError` si un substituant n'est ni dans *mapping*, ni dans *kwds*, c'est le nom du substituant inchangé qui apparaît dans la chaîne finale. Également, à l'inverse de `substitute()`, toute autre apparition de `$` renverra simplement `$` au lieu de lever une exception `ValueError`.

Bien que d'autres exceptions peuvent toujours être levées, cette méthode est dite sûre car elle tente de toujours renvoyer une chaîne utilisable au lieu de lever une exception. Ceci dit, `safe_substitute()` est tout sauf sûre car elle ignore silencieusement toute malformation dans le *template* qui contient des délimiteurs fantômes, des accolades non fermées, ou des substituants qui ne sont pas des identificateurs Python valides.

is_valid()

Returns false if the template has invalid placeholders that will cause *substitute()* to raise *ValueError*.

Ajouté dans la version 3.11.

get_identifiers()

Returns a list of the valid identifiers in the template, in the order they first appear, ignoring any invalid identifiers.

Ajouté dans la version 3.11.

Les instances de la classe *Template* fournissent également un attribut public :

template

C'est l'objet *template* passé comme argument au constructeur. En général, vous ne devriez pas le changer, mais un accès en lecture-seule n'est pas possible à fournir.

Voici un exemple de comment utiliser un *Template* :

```
>>> from string import Template
>>> s = Template('$who likes $what')
>>> s.substitute(who='tim', what='kung pao')
'tim likes kung pao'
>>> d = dict(who='tim')
>>> Template('Give $who $100').substitute(d)
Traceback (most recent call last):
...
ValueError: Invalid placeholder in string: line 1, col 11
>>> Template('$who likes $what').substitute(d)
Traceback (most recent call last):
...
KeyError: 'what'
>>> Template('$who likes $what').safe_substitute(d)
'tim likes $what'
```

Usage avancé : vous pouvez faire dériver vos sous-classes de *Template* pour personnaliser la syntaxe des substituants, les caractères délimiteurs, ou l'entièreté de l'expression rationnelle utilisée pour analyser les chaînes *templates*. Pour faire cela, vous pouvez redéfinir les attributs suivants :

— *delimiter* — La chaîne littérale décrivant le délimiteur pour introduire un substituant. Sa valeur par défaut est `$`. Notez qu'elle ne doit *pas* être une expression rationnelle, puisque l'implémentation appelle *re.escape()* sur cette chaîne si nécessaire. Notez aussi que le délimiteur ne peut pas être changé après la création de la classe.

— *idpattern* — L'expression rationnelle décrivant le motif pour les substituants non entourés d'accolades. La valeur par défaut de cette expression rationnelle est `(?a:[_a-z][_a-z0-9]*)`. Si *idpattern* est donné et *braceidpattern* est `None`, ce motif est aussi utilisé pour les marqueurs entre accolades.

> **Note**
>
> Puisque par défaut *flags* vaut `re.IGNORECASE`, des caractères *non-ASCII* peuvent correspondre au motif `[a-z]`. C'est pourquoi on utilise une option locale `a` ici.

Modifié dans la version 3.7 : *braceidpattern* peut être utilisé pour définir des motifs des motifs différents suivant qu'ils sont à l'intérieur ou à l'extérieur des accolades.

— *braceidpattern* — Similaire à *idpattern* mais décrit le motif quand il est placé entre accolades. La valeur par défaut est `None` ce qui signifie que seul *idpattern* est pris en compte (le motif est le même, qu'il soit à l'intérieur d'accolades ou non). S'il est donné, cela vous permet de définir des motifs entre accolades différents des motifs sans accolades.

Ajouté dans la version 3.7.

— *flags* — L'indicateur d'expression rationnelle qui sera appliqué lors de la compilation de l'expression rationnelle pour reconnaître les substitutions. La valeur par défaut est `re.IGNORECASE`. Notez que `re.VERBOSE` sera toujours ajouté à l'indicateur. Donc, un *idpattern* personnalisé doit suivre les conventions des expressions rationnelles *verbose*.

Ajouté dans la version 3.2.

Également, vous pouvez fournir le motif d'expression rationnelle en entier en redéfinissant l'attribut *pattern*. Si vous faites cela, la valeur doit être un objet 'expression rationnelle' avec quatre groupes de capture de noms. Les groupes de capture correspondent aux règles données au-dessus, ainsi qu'à la règle du substituant invalide :

— *escaped* — Ce groupe lie les séquences échappées (par exemple `$$`) dans le motif par défaut.
— *named* — Ce groupe lie les substituants non entourés d'accolades ; il ne devrait pas inclure le délimiteur dans le groupe de capture.
— *braced* — Ce groupe lie le nom entouré d'accolades ; il ne devrait inclure ni le délimiteur, ni les accolades dans le groupe de capture.
— *invalid* — Ce groupe lie tout autre motif de délimitation (habituellement, un seul délimiteur) et il devrait apparaître en dernier dans l'expression rationnelle.

The methods on this class will raise `ValueError` if the pattern matches the template without one of these named groups matching.

6.1.5 Fonctions d'assistance

`string.`**`capwords`**(*s*, *sep=None*)

Divise l'argument en mots en utilisant `str.split()`, capitalise chaque mot en utilisant `str.capitalize()` et assemble les mots capitalisés en utilisant `str.join()`. Si le second argument optionnel *sep* est absent ou vaut `None`, les séquences de caractères blancs sont remplacées par un seul espace et les espaces débutant et finissant la chaîne sont retirés. Sinon, *sep* et utilisé pour séparer et ré-assembler les mots.

6.2 `re` --- Regular expression operations

Code source : Lib/re/

Ce module fournit des opérations sur les expressions rationnelles similaires à celles que l'on trouve dans Perl.

Both patterns and strings to be searched can be Unicode strings (`str`) as well as 8-bit strings (`bytes`). However, Unicode strings and 8-bit strings cannot be mixed : that is, you cannot match a Unicode string with a bytes pattern or vice-versa ; similarly, when asking for a substitution, the replacement string must be of the same type as both the pattern and the search string.

Regular expressions use the backslash character (`'\'`) to indicate special forms or to allow special characters to be used without invoking their special meaning. This collides with Python's usage of the same character for the same purpose in string literals ; for example, to match a literal backslash, one might have to write `'\\\\'` as the pattern string, because the regular expression must be `\\`, and each backslash must be expressed as `\\` inside a regular Python string literal. Also, please note that any invalid escape sequences in Python's usage of the backslash in string literals now generate a `SyntaxWarning` and in the future this will become a `SyntaxError`. This behaviour will happen even if it is a valid escape sequence for a regular expression.

La solution est d'utiliser la notation des chaînes brutes en Python pour les expressions rationnelles ; Les *backslashs* ne provoquent aucun traitement spécifique dans les chaînes littérales préfixées par `'r'`. Ainsi, `r"\n"` est une chaîne de deux caractères contenant `'\'` et `'n'`, tandis que `"\n"` est une chaîne contenant un unique caractère : un saut de ligne. Généralement, les motifs seront exprimés en Python à l'aide de chaînes brutes.

Il est important de noter que la plupart des opérations sur les expressions rationnelles sont disponibles comme fonctions au niveau du module et comme méthodes des *expressions rationnelles compilées*. Les fonctions sont des raccourcis qui ne vous obligent pas à d'abord compiler un objet *regex*, mais auxquelles manquent certains paramètres de configuration fine.

Voir aussi

The third-party regex module, which has an API compatible with the standard library `re` module, but offers additional functionality and a more thorough Unicode support.

6.2.1 Syntaxe des expressions rationnelles

Une expression rationnelle (*regular expression* ou *RE*) spécifie un ensemble de chaînes de caractères qui lui correspondent ; les fonctions de ce module vous permettent de vérifier si une chaîne particulière correspond à une expression rationnelle donnée (ou si un expression rationnelle donnée correspond à une chaîne particulière, ce qui revient à la même chose).

Les expressions rationnelles peuvent être concaténées pour former de nouvelles expressions : si *A* et *B* sont deux expressions rationnelles, alors *AB* est aussi une expression rationnelle. En général, si une chaîne *p* valide *A* et qu'une autre chaîne *q* valide *B*, la chaîne *pq* validera AB. Cela est vrai tant que *A* et *B* ne contiennent pas d'opérations de priorité ; de conditions de frontière entre *A* et *B* ; ou de références vers des groupes numérotés. Ainsi, des expressions complexes peuvent facilement être construites depuis de plus simples expressions primitives comme celles décrites ici. Pour plus de détails sur la théorie et l'implémentation des expressions rationnelles, consultez le livre de Friedl [Frie09], ou à peu près n'importe quel livre dédié à la construction de compilateurs.

Une brève explication sur le format des expressions rationnelles suit. Pour de plus amples informations et une présentation plus simple, référez-vous au regex-howto.

Les expressions rationnelles peuvent contenir à la fois des caractères spéciaux et ordinaires. Les plus ordinaires, comme `'A'`, `'a'` ou `'0'` sont les expressions rationnelles les plus simples : elles correspondent simplement à elles-mêmes. Vous pouvez concaténer des caractères ordinaires, ainsi `last` correspond à la chaîne `'last'`. (Dans la suite de cette section, nous écrirons les expressions rationnelles dans `ce style spécifique`, généralement sans guillemets, et les chaînes à tester `'entourées de simples guillemets'`.)

Certains caractères, comme `'|'` ou `'('`, sont spéciaux. Des caractères spéciaux peuvent aussi exister pour les classes de caractères ordinaires, ou affecter comment les expressions rationnelles autour d'eux seront interprétées.

Les caractères de répétition ou quantificateurs (`*`, `+`, `?`, `{m,n}`, etc.) ne peuvent être directement imbriqués. Cela empêche l'ambiguïté avec le suffixe modificateur non gourmand `?` et avec les autres modificateurs dans d'autres implémentations. Pour appliquer une seconde répétition à une première, des parenthèses peuvent être utilisées. Par exemple, l'expression `(?:a{6})*` valide toutes les chaînes composées d'un nombre de caractères `'a'` multiple de six.

Les caractères spéciaux sont :

.

(Dot.) In the default mode, this matches any character except a newline. If the `DOTALL` flag has been specified, this matches any character including a newline. `(?s:.)` matches any character regardless of flags.

^

(Accent circonflexe.) Valide le début d'une chaîne de caractères, ainsi que ce qui suit chaque saut de ligne en mode `MULTILINE`.

$

Valide la fin d'une chaîne de caractères, ou juste avant le saut de ligne à la fin de la chaîne, ainsi qu'avant chaque saut de ligne en mode `MULTILINE`. `foo` valide à la fois *foo* et *foobar*, tandis que l'expression rationnelle `foo$` ne correspond qu'à `'foo'`. Plus intéressant, chercher `foo.$` dans `'foo1\nfoo2\n'` trouve normalement `'foo2'`, mais `'foo1'` en mode `MULTILINE` ; chercher un simple `$` dans `'foo\n'` trouvera deux correspondances (vides) : une juste avant le saut de ligne, et une à la fin de la chaîne.

*****

Fait valider par l'expression rationnelle résultante 0 répétition ou plus de l'expression qui précède, avec autant de répétitions que possible. `ab*` validera 'a', 'ab' ou 'a' suivi de n'importe quel nombre de 'b'.

+

Fait valider par l'expression rationnelle résultante 1 répétition ou plus de l'expression qui précède. `ab+` validera 'a' suivi de n'importe quel nombre non nul de 'b' ; cela ne validera pas la chaîne 'a'.

?

Fait valider par l'expression rationnelle résultante 0 ou 1 répétition de l'expression qui précède. `ab?` correspondra à 'a' ou 'ab'.

***?, +?, ??**

Les quantificateurs `'*'`, `'+'` et `'?'` sont tous *greedy* (gourmands) ; ils valident autant de texte que possible. Parfois ce comportement n'est pas désiré ; si l'expression rationnelle `<.*>` est testée avec la chaîne `'<a> b <c>'`, cela correspondra à la chaîne entière, et non juste à `'<a>'`. Ajouter `?` derrière le quantificateur lui fait réaliser l'opération de façon *non-greedy* (ou *minimal*) ; le *moins* de caractères possibles seront validés. Utiliser l'expression rationnelle `<.*?>` validera uniquement `'<a>'`.

***+, ++, ?+**

Like the `'*'`, `'+'`, and `'?'` quantifiers, those where `'+'` is appended also match as many times as possible. However, unlike the true greedy quantifiers, these do not allow back-tracking when the expression following it fails to match. These are known as *possessive* quantifiers. For example, `a*a` will match `'aaaa'` because the `a*` will match all 4 `'a'`s, but, when the final `'a'` is encountered, the expression is backtracked so that in the end the `a*` ends up matching 3 `'a'`s total, and the fourth `'a'` is matched by the final `'a'`. However, when `a*+a` is used to match `'aaaa'`, the `a*+` will match all 4 `'a'`, but when the final `'a'` fails to find any more characters to match, the expression cannot be backtracked and will thus fail to match. `x*+`, `x++` and `x?+` are equivalent to `(?>x*)`, `(?>x+)` and `(?>x?)` correspondingly.

Ajouté dans la version 3.11.

{m}

Spécifie qu'exactement *m* copies de l'expression rationnelle qui précède devront être validées ; un nombre plus faible de correspondances empêche l'expression entière de correspondre. Par exemple, `a{6}` correspondra exactement à six caractères `'a'`, mais pas à cinq.

{m,n}

Fait valider par l'expression rationnelle résultante entre *m* et *n* répétitions de l'expression qui précède, cherchant à en valider le plus possible. Par exemple, `a{3,5}` validera entre 3 et 5 caractères `'a'`. Omettre *m* revient à spécifier 0 comme borne inférieure, et omettre *n* à avoir une borne supérieure infinie. Par exemple, `a{4,}b` correspondra à `'aaaab'` ou à un millier de caractères `'a'` suivis d'un `'b'`, mais pas à `'aaab'`. La virgule ne doit pas être omise, auquel cas le modificateur serait confondu avec la forme décrite précédemment.

{m,n}?

Fait valider l'expression rationnelle résultante entre *m* et *n* répétitions de l'expression qui précède, cherchant à en valider le moins possible. Il s'agit de la version non gourmande du précédent quantificateur. Par exemple, dans la chaîne de 6 caractères `'aaaaaa'`, `a{3,5}` trouvera 5 caractères `'a'`, alors que `a{3,5}?` n'en trouvera que 3.

{m,n}+

Causes the resulting RE to match from *m* to *n* repetitions of the preceding RE, attempting to match as many repetitions as possible *without* establishing any backtracking points. This is the possessive version of the quantifier above. For example, on the 6-character string `'aaaaaa'`, `a{3,5}+aa` attempt to match 5 `'a'` characters, then, requiring 2 more `'a'`s, will need more characters than available and thus fail, while `a{3,5}aa` will match with `a{3,5}` capturing 5, then 4 `'a'`s by backtracking and then the final 2 `'a'`s are matched by the final `aa` in the pattern. `x{m,n}+` is equivalent to `(?>x{m,n})`.

Ajouté dans la version 3.11.

****

Échappe les caractères spéciaux (permettant d'identifier des caractères comme `'*'`, `'?'` et autres) ou signale une séquence spéciale ; les séquences spéciales sont décrites ci-dessous.

Si vous n'utilisez pas de chaînes brutes pour exprimer le motif, souvenez-vous que Python utilise aussi le *backslash* comme une séquence d'échappement dans les chaînes littérales ; si la séquence d'échappement n'est

pas reconnue par l'interpréteur Python, le *backslash* et les caractères qui le suivent sont inclus dans la chaîne renvoyée. Cependant, si Python reconnait la séquence, le *backslash* doit être doublé (pour ne plus être reconnu). C'est assez compliqué et difficile à comprendre, c'est pourquoi il est hautement recommandé d'utiliser des chaînes brutes pour tout sauf les expressions les plus simples.

[]

Utilisé pour indiquer un ensemble de caractères. Dans un ensemble :

- Les caractères peuvent être listés individuellement, e.g. `[amk]` correspondra à `'a'`, `'m'` ou `'k'`.
- Des intervalles de caractères peuvent être indiqués en donnant deux caractères et les séparant par un `'-'`, par exemple `[a-z]` correspondra à toute lettre minuscule *ASCII*, `[0-5][0-9]` à tous nombres de deux chiffres entre `00` et `59`, et `[0-9A-Fa-f]` correspondra à n'importe quel chiffre hexadécimal. Si `'-'` est échappé (`[a\-z]`) ou s'il est placé comme premier ou dernier caractère (e.g. `[-a]` ou `[a-]`), il correspondra à un `'-'` littéral.
- Les caractères spéciaux perdent leur sens à l'intérieur des ensembles. Par exemple, `[(+*)]` validera chacun des caractères littéraux `'('`, `'+'`, `'*'` ou `')'`.
- Character classes such as `\w` or `\S` (defined below) are also accepted inside a set, although the characters they match depend on the *flags* used.
- Les caractères qui ne sont pas dans un intervalle peuvent être trouvés avec l'ensemble complémentaire (*complementing*). Si le premier caractère de l'ensemble est `'^'`, tous les caractères qui *ne sont pas* dans l'ensemble seront validés. Par exemple, `[^5]` correspondra à tout caractère autre que `'5'` et `[^^]` validera n'importe quel caractère excepté `'^'`. `^` n'a pas de sens particulier s'il n'est pas le premier caractère de l'ensemble.
- To match a literal `']'` inside a set, precede it with a backslash, or place it at the beginning of the set. For example, both `[()[\]{}]` and `[]()[{}]` will match a right bracket, as well as left bracket, braces, and parentheses.
- La gestion des ensembles inclus l'un dans l'autre et les opérations d'ensemble comme dans Unicode Technical Standard #18 pourrait être ajoutée par la suite. Ceci changerait la syntaxe, donc pour faciliter ce changement, une exception `FutureWarning` sera levée dans les cas ambigus pour le moment. Ceci inclut les ensembles commençant avec le caractère `'['` ou contenant les séquences de caractères `'--'`, `'&&'`, `'~~'` et `'||'`. Pour éviter un message d'avertissement, échapper les séquences avec le caractère antislash (`"\"`).

Modifié dans la version 3.7 : L'exception `FutureWarning` est levée si un ensemble de caractères contient une construction dont la sémantique changera dans le futur.

|

`A|B`, où *A* et *B* peuvent être deux expressions rationnelles arbitraires, crée une expression rationnelle qui validera soit *A* soit *B*. Un nombre arbitraire d'expressions peuvent être séparées de cette façon par des `'|'`. Cela peut aussi être utilisé au sein de groupes (voir ci-dessous). Quand une chaîne cible est analysée, les expressions séparées par `'|'` sont essayées de la gauche vers la droite. Quand un motif correspond complètement, cette branche est acceptée. Cela signifie qu'une fois que *A* correspond, *B* ne sera pas testée plus loin, même si elle pourrait provoquer une plus ample correspondance. En d'autres termes, l'opérateur `'|'` n'est jamais gourmand. Pour valider un `'|'` littéral, utilisez `\|`, ou enveloppez-le dans une classe de caractères, comme `[|]`.

(...)

Valide n'importe quelle expression rationnelle comprise entre les parenthèses, et indique le début et la fin d'un groupe ; le contenu d'un groupe peut être récupéré après qu'une analyse a été effectuée et peut être réutilisé plus loin dans la chaîne avec une séquence spéciale `\number`, décrite ci-dessous. Pour écrire des `'('` ou `')'` littéraux, utilisez `\(` ou `\)`, ou enveloppez-les dans une classe de caractères : `[(]`, `[)]`.

(?...)

Il s'agit d'une notation pour les extensions (un `'?'` suivant une `'('` n'a pas de sens autrement). Le premier caractère après le `'?'` détermine quel sens donner à l'expression. Les extensions ne créent généralement pas de nouveaux groupes ; `(?P<name>...)` est la seule exception à la règle. Retrouvez ci-dessous la liste des extensions actuellement supportées.

(?aiLmsux)

(One or more letters from the set `'a'`, `'i'`, `'L'`, `'m'`, `'s'`, `'u'`, `'x'`.) The group matches the empty string ; the letters set the corresponding flags for the entire regular expression :

— *re.A* (ASCII-only matching)
— *re.I* (ignore case)
— *re.L* (locale dependent)
— *re.M* (multi-line)
— *re.S* (dot matches all)
— *re.U* (Unicode matching)
— *re.X* (verbose)

(The flags are described in *Contenu du module*.) This is useful if you wish to include the flags as part of the regular expression, instead of passing a *flag* argument to the *re.compile()* function. Flags should be used first in the expression string.

Modifié dans la version 3.11 : Cette construction ne peut être utilisée qu'au début d'une chaîne de caractères.

(?:...)

Une version sans capture des parenthèses habituelles. Valide n'importe quelle expression rationnelle à l'intérieur des parenthèses, mais la sous-chaîne correspondant au groupe *ne peut pas* être récupérée après l'analyse ou être référencée plus loin dans le motif.

(?aiLmsux-imsx:...)

(Zero or more letters from the set `'a'`, `'i'`, `'L'`, `'m'`, `'s'`, `'u'`, `'x'`, optionally followed by `'-'` followed by one or more letters from the `'i'`, `'m'`, `'s'`, `'x'`.) The letters set or remove the corresponding flags for the part of the expression :

— *re.A* (ASCII-only matching)
— *re.I* (ignore case)
— *re.L* (locale dependent)
— *re.M* (multi-line)
— *re.S* (dot matches all)
— *re.U* (Unicode matching)
— *re.X* (verbose)

(The flags are described in *Contenu du module*.)

The letters `'a'`, `'L'` and `'u'` are mutually exclusive when used as inline flags, so they can't be combined or follow `'-'`. Instead, when one of them appears in an inline group, it overrides the matching mode in the enclosing group. In Unicode patterns `(?a:...)` switches to ASCII-only matching, and `(?u:...)` switches to Unicode matching (default). In bytes patterns `(?L:...)` switches to locale dependent matching, and `(?a:...)` switches to ASCII-only matching (default). This override is only in effect for the narrow inline group, and the original matching mode is restored outside of the group.

Ajouté dans la version 3.6.

Modifié dans la version 3.7 : Les lettres `'a'`, `'L'` et `'u'` peuvent aussi être utilisées dans un groupe.

(?>...)

Attempts to match `...` as if it was a separate regular expression, and if successful, continues to match the rest of the pattern following it. If the subsequent pattern fails to match, the stack can only be unwound to a point *before* the `(?>...)` because once exited, the expression, known as an *atomic group*, has thrown away all stack points within itself. Thus, `(?>.*).` would never match anything because first the `.*` would match all characters possible, then, having nothing left to match, the final `.` would fail to match. Since there are no stack points saved in the Atomic Group, and there is no stack point before it, the entire expression would thus fail to match.

Ajouté dans la version 3.11.

(?P<name>...)

Similar to regular parentheses, but the substring matched by the group is accessible via the symbolic group name *name*. Group names must be valid Python identifiers, and in *bytes* patterns they can only contain bytes in the ASCII range. Each group name must be defined only once within a regular expression. A symbolic group is also a numbered group, just as if the group were not named.

Les groupes nommés peuvent être référencés dans trois contextes. Si le motif est `(?P<quote>['"]).*?(?P=quote)` (c.-à-d. correspondant à une chaîne entourée de guillemets simples ou doubles) :

Contexte de référence au groupe *quote*	Manières de le référencer
lui-même dans le même motif	— `(?P=quote)` (comme vu) — `\1`
en analysant l'objet résultat *m*	— `m.group('quote')` — `m.end('quote')` (etc.)
dans une chaîne passée à l'argument *repl* de `re.sub()`	— `\g<quote>` — `\g<1>` — `\1`

Modifié dans la version 3.12 : In `bytes` patterns, group *name* can only contain bytes in the ASCII range (`b'\x00'-b'\x7f'`).

`(?P=name)`

Une référence arrière à un groupe nommé ; elle correspond à n'importe quel texte validé plus tôt par le groupe nommé *name*.

`(?#...)`

Un commentaire ; le contenu des parenthèses est simplement ignoré.

`(?=...)`

Valide si `...` valide la suite, mais ne consomme rien de la chaîne. On appelle cela une assertion *lookahead*. Par exemple, `Isaac (?=Asimov)` correspondra à la chaîne `'Isaac '` seulement si elle est suivie par `'Asimov'`.

`(?!...)`

Valide si `...` ne valide pas la suite. C'est une assertion *negative lookahead*. Par exemple, `Isaac (?!Asimov)` correspondra à la chaîne `'Isaac '` seulement si elle *n'est pas* suivie par `'Asimov'`.

`(?<=...)`

Valide si la position courante dans la chaîne est précédée par une correspondance sur `...` qui se termine à la position courante. On appelle cela une *positive lookbehind assertion*. `(?<=abc)def` cherchera une correspondance dans `'abcdef'`, puisque le *lookbehind** mettra de côté 3 caractères et vérifiera que le motif contenu correspond. Le motif ne devra correspondre qu'à des chaînes de taille fixe, cela veut dire que `abc` ou `a|b` sont autorisées, mais pas `a*` ou `a{3,4}`. Notez que les motifs qui commencent par des assertions *lookbehind* positives ne peuvent pas correspondre au début de la chaîne analysée ; vous préférerez sûrement utiliser la fonction `search()` plutôt que la fonction `match()` :

```
>>> import re
>>> m = re.search('(?<=abc)def', 'abcdef')
>>> m.group(0)
'def'
```

Cet exemple recherche un mot suivi d'un trait d'union :

```
>>> m = re.search(r'(?<=-)\w+', 'spam-egg')
>>> m.group(0)
'egg'
```

Modifié dans la version 3.5 : Ajout du support des références aux groupes de taille fixe.

(?<!...)

Valide si la position courante dans la chaîne n'est pas précédée par une correspondance sur `...`. On appelle cela une *negative lookbehind assertion*. À la manière des assertions *lookbehind* positives, le motif contenu ne peut que correspondre à des chaînes de taille fixe. Les motifs débutant par une assertion *lookbehind* négative peuvent correspondre au début de la chaîne analysée.

(?(id/name)yes-pattern|no-pattern)

Essaiera de faire la correspondance avec `yes-pattern` si le groupe indiqué par *id* ou *name* existe, et avec `no-pattern` s'il n'existe pas. `no-pattern` est optionnel et peut être omis. Par exemple, `(<)?(\w+@\w+(?:\.\w+)+)(?(1)>|$)` est un motif simpliste pour identifier une adresse courriel, qui validera `'<user@host.com>'` ainsi que `'user@host.com'` mais pas `'<user@host.com'` ni `'user@host.com>'`.

Modifié dans la version 3.12 : Group *id* can only contain ASCII digits. In `bytes` patterns, group *name* can only contain bytes in the ASCII range (`b'\x00'-b'\x7f'`).

Les séquences spéciales sont composées de `'\'` et d'un caractère de la liste qui suit. Si le caractère ordinaire n'est pas un chiffre *ASCII* ou une lettre *ASCII*, alors l'expression rationnelle résultante validera le second caractère de la séquence. Par exemple, `\$` correspond au caractère `'$'`.

\number

Correspond au contenu du groupe du même nombre. Les groupes sont numérotés à partir de 1. Par exemple, `(.+) \1` correspond à `'the the'` ou `'55 55'`, mais pas à `'thethe'` (notez l'espace après le groupe). Cette séquence spéciale ne peut être utilisée que pour faire référence aux 99 premiers groupes. Si le premier chiffre de *number* est 0, ou si *number* est un nombre octal de 3 chiffres, il ne sera pas interprété comme une référence à un groupe, mais comme le caractère à la valeur octale *number*. À l'intérieur des `'['` et `']'` d'une classe de caractères, tous les échappements numériques sont traités comme des caractères.

\A

Correspond uniquement au début d'une chaîne de caractères.

\b

Matches the empty string, but only at the beginning or end of a word. A word is defined as a sequence of word characters. Note that formally, `\b` is defined as the boundary between a `\w` and a `\W` character (or vice versa), or between `\w` and the beginning or end of the string. This means that `r'\bat\b'` matches `'at'`, `'at.'`, `'(at)'`, and `'as at ay'` but not `'attempt'` or `'atlas'`.

The default word characters in Unicode (str) patterns are Unicode alphanumerics and the underscore, but this can be changed by using the `ASCII` flag. Word boundaries are determined by the current locale if the `LOCALE` flag is used.

> **Note**
>
> Inside a character range, `\b` represents the backspace character, for compatibility with Python's string literals.

\B

Matches the empty string, but only when it is *not* at the beginning or end of a word. This means that `r'at\B'` matches `'athens'`, `'atom'`, `'attorney'`, but not `'at'`, `'at.'`, or `'at!'`. `\B` is the opposite of `\b`, so word characters in Unicode (str) patterns are Unicode alphanumerics or the underscore, although this can be changed by using the `ASCII` flag. Word boundaries are determined by the current locale if the `LOCALE` flag is used.

Modifié dans la version 3.14 : `\B` now matches empty input string.

\d

Pour les motifs Unicode (*str*) :

Matches any Unicode decimal digit (that is, any character in Unicode character category [Nd]). This includes `[0-9]`, and also many other digit characters.

Matches `[0-9]` if the *ASCII* flag is used.

Pour les motifs 8-bits (*bytes*) :

Matches any decimal digit in the ASCII character set ; this is equivalent to `[0-9]`.

\D

Matches any character which is not a decimal digit. This is the opposite of `\d`.

Matches `[^0-9]` if the *ASCII* flag is used.

\s

Pour les motifs Unicode (*str*) :

Matches Unicode whitespace characters (as defined by *str.isspace()*). This includes `[ \t\n\r\f\v]`, and also many other characters, for example the non-breaking spaces mandated by typography rules in many languages.

Matches `[ \t\n\r\f\v]` if the *ASCII* flag is used.

Pour les motifs 8-bits (*bytes*) :

Valide les caractères considérés comme des espacements dans la table ASCII ; équivalent à `[ \t\n\r\f\v]`.

\S

Matches any character which is not a whitespace character. This is the opposite of `\s`.

Matches `[^ \t\n\r\f\v]` if the *ASCII* flag is used.

\w

Pour les motifs Unicode (*str*) :

Matches Unicode word characters ; this includes all Unicode alphanumeric characters (as defined by *str.isalnum()*), as well as the underscore (_).

Matches `[a-zA-Z0-9_]` if the *ASCII* flag is used.

Pour les motifs 8-bits (*bytes*) :

Matches characters considered alphanumeric in the ASCII character set ; this is equivalent to `[a-zA-Z0-9_]`. If the *LOCALE* flag is used, matches characters considered alphanumeric in the current locale and the underscore.

\W

Matches any character which is not a word character. This is the opposite of `\w`. By default, matches non-underscore (_) characters for which *str.isalnum()* returns `False`.

Matches `[^a-zA-Z0-9_]` if the *ASCII* flag is used.

If the *LOCALE* flag is used, matches characters which are neither alphanumeric in the current locale nor the underscore.

\Z

Correspond uniquement à la fin d'une chaîne de caractères.

Most of the escape sequences supported by Python string literals are also accepted by the regular expression parser :

```
\a      \b      \f      \n
\N      \r      \t      \u
\U      \v      \x      \\
```

(Notez que `\b` est utilisé pour représenter les bornes d'un mot, et signifie « *retour arrière* » uniquement à l'intérieur d'une classe de caractères)

`'\u'`, `'\U'`, and `'\N'` escape sequences are only recognized in Unicode (str) patterns. In bytes patterns they are errors. Unknown escapes of ASCII letters are reserved for future use and treated as errors.

Les séquences octales d'échappement sont incluses dans une forme limitée. Si le premier chiffre est un 0, ou s'il y a trois chiffres octaux, la séquence est considérée comme octale. Autrement, il s'agit d'une référence vers un groupe. Comme pour les chaînes littérales, les séquences octales ne font jamais plus de 3 caractères de long.

Modifié dans la version 3.3 : Les séquences d'échappement `'\u'` et `'\U'` ont été ajoutées.

Modifié dans la version 3.6 : Les séquences inconnues composées de `'\'` et d'une lettre ASCII sont maintenant des erreurs.

Modifié dans la version 3.8 : The `'\N{name}'` escape sequence has been added. As in string literals, it expands to the named Unicode character (e.g. `'\N{EM DASH}'`).

6.2.2 Contenu du module

Le module définit plusieurs fonctions, constantes, et une exception. Certaines fonctions sont des versions simplifiées des méthodes plus complètes des expressions rationnelles compilées. La plupart des applications non triviales utilisent toujours la version compilée.

Flags

Modifié dans la version 3.6 : Les constantes d'options sont maintenant des instances de *RegexFlag*, sous-classe de *enum.IntFlag*.

class re.**RegexFlag**

An *enum.IntFlag* class containing the regex options listed below.

Ajouté dans la version 3.11 : - added to `__all__`

re.**A**

re.**ASCII**

Make `\w`, `\W`, `\b`, `\B`, `\d`, `\D`, `\s` and `\S` perform ASCII-only matching instead of full Unicode matching. This is only meaningful for Unicode (str) patterns, and is ignored for bytes patterns.

Corresponds to the inline flag `(?a)`.

> **Note**
>
> The *U* flag still exists for backward compatibility, but is redundant in Python 3 since matches are Unicode by default for `str` patterns, and Unicode matching isn't allowed for bytes patterns. *UNICODE* and the inline flag `(?u)` are similarly redundant.

re.**DEBUG**

Display debug information about compiled expression.

No corresponding inline flag.

re.**I**

re.**IGNORECASE**

Perform case-insensitive matching; expressions like `[A-Z]` will also match lowercase letters. Full Unicode matching (such as Ü matching ü) also works unless the *ASCII* flag is used to disable non-ASCII matches. The current locale does not change the effect of this flag unless the *LOCALE* flag is also used.

Corresponds to the inline flag `(?i)`.

Note that when the Unicode patterns `[a-z]` or `[A-Z]` are used in combination with the *IGNORECASE* flag, they will match the 52 ASCII letters and 4 additional non-ASCII letters : 'İ' (U+0130, Latin capital letter I with

dot above), 'ı' (U+0131, Latin small letter dotless i), 'ſ' (U+017F, Latin small letter long s) and 'K' (U+212A, Kelvin sign). If the *ASCII* flag is used, only letters 'a' to 'z' and 'A' to 'Z' are matched.

re.**L**

re.**LOCALE**

Make `\w`, `\W`, `\b`, `\B` and case-insensitive matching dependent on the current locale. This flag can be used only with bytes patterns.

Corresponds to the inline flag `(?L)`.

> **Avertissement**
>
> This flag is discouraged ; consider Unicode matching instead. The locale mechanism is very unreliable as it only handles one "culture" at a time and only works with 8-bit locales. Unicode matching is enabled by default for Unicode (str) patterns and it is able to handle different locales and languages.

Modifié dans la version 3.6 : *LOCALE* can be used only with bytes patterns and is not compatible with *ASCII*.

Modifié dans la version 3.7 : Compiled regular expression objects with the *LOCALE* flag no longer depend on the locale at compile time. Only the locale at matching time affects the result of matching.

re.**M**

re.**MULTILINE**

When specified, the pattern character `'^'` matches at the beginning of the string and at the beginning of each line (immediately following each newline) ; and the pattern character `'$'` matches at the end of the string and at the end of each line (immediately preceding each newline). By default, `'^'` matches only at the beginning of the string, and `'$'` only at the end of the string and immediately before the newline (if any) at the end of the string.

Corresponds to the inline flag `(?m)`.

re.**NOFLAG**

Indicates no flag being applied, the value is `0`. This flag may be used as a default value for a function keyword argument or as a base value that will be conditionally ORed with other flags. Example of use as a default value :

```
def myfunc(text, flag=re.NOFLAG):
    return re.match(text, flag)
```

Ajouté dans la version 3.11.

re.**S**

re.**DOTALL**

Make the `'.'` special character match any character at all, including a newline ; without this flag, `'.'` will match anything *except* a newline.

Corresponds to the inline flag `(?s)`.

re.**U**

re.**UNICODE**

In Python 3, Unicode characters are matched by default for `str` patterns. This flag is therefore redundant with **no effect** and is only kept for backward compatibility.

See *ASCII* to restrict matching to ASCII characters instead.

re.**X**

re.**VERBOSE**

Cette option vous autorise à écrire des expressions rationnelles qui présentent mieux et sont plus lisibles en vous permettant de séparer visuellement les sections logiques du motif et d'ajouter des commentaires. Les caractères d'espacement à l'intérieur du motif sont ignorés, sauf à l'intérieur des classes de caractères ou quand ils sont précédés d'un *backslash* non échappé, ou dans des séquences comme `*?`, `(?:` ou `(?P<...>`.Par exemple, `(?`

`:` et `* ?` ne sont pas autorisés. Quand une ligne contient un `#` qui n'est ni dans une classe de caractères, ni précédé d'un *backslash* non échappé, tous les caractères depuis le `#` le plus à gauche jusqu'à la fin de la ligne sont ignorés.

Cela signifie que les deux expressions rationnelles suivantes qui valident un nombre décimal sont fonctionnellement égales :

```
a = re.compile(r"""\d +  # the integral part
                  \.    # the decimal point
                  \d *  # some fractional digits""", re.X)
b = re.compile(r"\d+\.\d*")
```

Correspond à l'option de groupe `(?x)`.

Fonctions

`re.`**`compile`**(*pattern*, *flags=0*)

Compile un motif vers une *expression rationnelle* compilée, dont les méthodes `match()` et `search()`, décrites ci-dessous, peuvent être utilisées pour analyser des textes.

The expression's behaviour can be modified by specifying a *flags* value. Values can be any of the *flags* variables, combined using bitwise OR (the `|` operator).

La séquence

```
prog = re.compile(pattern)
result = prog.match(string)
```

est équivalente à

```
result = re.match(pattern, string)
```

mais utiliser `re.compile()` et sauvegarder l'expression rationnelle renvoyée pour la réutiliser est plus efficace quand l'expression est amenée à être utilisée plusieurs fois dans un même programme.

> **Note**
>
> Les versions compilées des motifs les plus récents passés à `re.compile()` et autres fonctions d'analyse du module sont mises en cache, ainsi les programmes qui n'utilisent que quelques expressions rationnelles en même temps n'ont pas à s'inquiéter de la compilation de ces expressions.

`re.`**`search`**(*pattern*, *string*, *flags=0*)

Scan through *string* looking for the first location where the regular expression *pattern* produces a match, and return a corresponding `Match`. Return `None` if no position in the string matches the pattern ; note that this is different from finding a zero-length match at some point in the string.

The expression's behaviour can be modified by specifying a *flags* value. Values can be any of the *flags* variables, combined using bitwise OR (the `|` operator).

`re.`**`match`**(*pattern*, *string*, *flags=0*)

If zero or more characters at the beginning of *string* match the regular expression *pattern*, return a corresponding `Match`. Return `None` if the string does not match the pattern ; note that this is different from a zero-length match.

Notez que même en mode `MULTILINE`, `re.match()` ne validera qu'au début de la chaîne et non au début de chaque ligne.

Si vous voulez trouver une correspondance n'importe où dans *string*, utilisez plutôt `search()` (voir aussi *Comparaison de search() et match()*).

The expression's behaviour can be modified by specifying a *flags* value. Values can be any of the *flags* variables, combined using bitwise OR (the `|` operator).

`re.fullmatch`(*pattern*, *string*, *flags=0*)

If the whole *string* matches the regular expression *pattern*, return a corresponding `Match`. Return `None` if the string does not match the pattern ; note that this is different from a zero-length match.

The expression's behaviour can be modified by specifying a *flags* value. Values can be any of the *flags* variables, combined using bitwise OR (the `|` operator).

Ajouté dans la version 3.4.

`re.split`(*pattern*, *string*, *maxsplit=0*, *flags=0*)

Sépare *string* selon les occurrences de *pattern*. Si des parenthèses de capture sont utilisées dans *pattern*, alors les textes des groupes du motif sont aussi renvoyés comme éléments de la liste résultante. Si *maxsplit* est différent de zéro, il ne pourra y avoir plus de *maxsplit* séparations, et le reste de la chaîne sera renvoyé comme le dernier élément de la liste.

```
>>> re.split(r'\W+', 'Words, words, words.')
['Words', 'words', 'words', '']
>>> re.split(r'(\W+)', 'Words, words, words.')
['Words', ', ', 'words', ', ', 'words', '.', '']
>>> re.split(r'\W+', 'Words, words, words.', maxsplit=1)
['Words', 'words, words.']
>>> re.split('[a-f]+', '0a3B9', flags=re.IGNORECASE)
['0', '3', '9']
```

S'il y a des groupes de capture dans le séparateur et qu'ils trouvent une correspondance au début de la chaîne, le résultat commencera par une chaîne vide. La même chose se produit pour la fin de la chaîne :

```
>>> re.split(r'(\W+)', '...words, words...')
['', '...', 'words', ', ', 'words', '...', '']
```

De cette manière, les séparateurs sont toujours trouvés aux mêmes indices relatifs dans la liste résultante.

Les correspondances vides pour le motif scindent la chaîne de caractères seulement lorsqu'ils ne sont pas adjacents à une correspondance vide précédente.

```
>>> re.split(r'\b', 'Words, words, words.')
['', 'Words', ', ', 'words', ', ', 'words', '.']
>>> re.split(r'\W*', '...words...')
['', '', 'w', 'o', 'r', 'd', 's', '', '']
>>> re.split(r'(\W*)', '...words...')
['', '...', '', '', 'w', '', 'o', '', 'r', '', 'd', '', 's', '...', '', '', '']
```

The expression's behaviour can be modified by specifying a *flags* value. Values can be any of the *flags* variables, combined using bitwise OR (the `|` operator).

Modifié dans la version 3.1 : ajout de l'argument optionnel *flags*.

Modifié dans la version 3.7 : Gestion du découpage avec un motif qui pourrait correspondre à une chaine de caractère vide.

Obsolète depuis la version 3.13 : Passing *maxsplit* and *flags* as positional arguments is deprecated. In future Python versions they will be *keyword-only parameters*.

`re.findall`(*pattern*, *string*, *flags=0*)

Renvoie toutes les correspondances, sans chevauchements, entre le motif *pattern* et la chaîne *string*, comme une liste de chaînes ou de *n*-uplets. La chaîne *string* est examinée de gauche à droite, et les correspondances sont données dans cet ordre. Le résultat peut contenir des correspondances vides.

Le type du résultat dépend du nombre de groupes capturants dans le motif. S'il n'y en a pas, le résultat est une liste de sous-chaînes de caractères qui correspondent au motif. S'il y a exactement un groupe, le résultat est une liste constituée des sous-chaînes qui correspondaient à ce groupe pour chaque correspondance entre le motif et la chaîne. S'il y a plusieurs groupes, le résultat est formé de *n*-uplets avec les sous-chaînes correspondant aux différents groupes.

```
>>> re.findall(r'\bf[a-z]*', 'which foot or hand fell fastest')
['foot', 'fell', 'fastest']
>>> re.findall(r'(\w+)=(\d+)', 'set width=20 and height=10')
[('width', '20'), ('height', '10')]
```

The expression's behaviour can be modified by specifying a *flags* value. Values can be any of the *flags* variables, combined using bitwise OR (the `|` operator).

Modifié dans la version 3.7 : Les correspondances non vides peuvent maintenant démarrer juste après une correspondance vide précédente.

`re.`**`finditer`**(*pattern*, *string*, *flags=0*)

Return an *iterator* yielding `Match` objects over all non-overlapping matches for the RE *pattern* in *string*. The *string* is scanned left-to-right, and matches are returned in the order found. Empty matches are included in the result.

The expression's behaviour can be modified by specifying a *flags* value. Values can be any of the *flags* variables, combined using bitwise OR (the `|` operator).

Modifié dans la version 3.7 : Les correspondances non vides peuvent maintenant démarrer juste après une correspondance vide précédente.

`re.`**`sub`**(*pattern*, *repl*, *string*, *count=0*, *flags=0*)

Renvoie la chaîne obtenue en remplaçant les occurrences (sans chevauchement) les plus à gauche de *pattern* dans *string* par le remplacement *repl*. Si le motif n'est pas trouvé, *string* est renvoyée inchangée. *repl* peut être une chaîne de caractères ou une fonction ; si c'est une chaîne, toutes les séquences d'échappement qu'elle contient sont traduites. Ainsi, `\n` est convertie en un simple saut de ligne, `\r` en un retour chariot, et ainsi de suite. Les échappements inconnus de lettres ASCII sont réservés pour une utilisation future et sont considérés comme des erreurs. Les autres échappements tels que `\&` sont laissés intacts. Les références arrières, telles que `\6`, sont remplacées par la sous-chaîne correspondant au groupe 6 dans le motif. Par exemple :

```
>>> re.sub(r'def\s+([a-zA-Z_][a-zA-Z_0-9]*)\s*\(\s*\):',
...        r'static PyObject*\npy_\1(void)\n{',
...        'def myfunc():')
'static PyObject*\npy_myfunc(void)\n{'
```

If *repl* is a function, it is called for every non-overlapping occurrence of *pattern*. The function takes a single `Match` argument, and returns the replacement string. For example :

```
>>> def dashrepl(matchobj):
...     if matchobj.group(0) == '-': return ' '
...     else: return '-'
...
>>> re.sub('-{1,2}', dashrepl, 'pro----gram-files')
'pro--gram files'
>>> re.sub(r'\sAND\s', ' & ', 'Baked Beans And Spam', flags=re.IGNORECASE)
'Baked Beans & Spam'
```

The pattern may be a string or a `Pattern`.

L'argument optionnel *count* est le nombre maximum d'occurrences du motif à remplacer : *count* ne doit pas être un nombre négatif. Si omis ou nul, toutes les occurrences seront remplacées. Les correspondances vides avec le motif sont remplacées uniquement quand elles ne sont pas adjacentes à une précédente correspondance, ainsi `sub('x*', '-', 'abxd')` renvoie `'-a-b--d-'`.

Dans les arguments *repl* de type *string*, en plus des séquences d'échappement et références arrières décrites au-dessus, `\g<name>` utilisera la sous-chaîne correspondant au groupe nommé `name`, comme défini par la syntaxe `(?P<name>...)`. `\g<number>` utilise le groupe numéroté associé ; `\g<2>` est ainsi équivalent à `\2`, mais n'est pas ambigu dans un remplacement tel que `\g<2>0`. `\20` serait interprété comme une référence au groupe 20, et non une référence au groupe 2 suivie par un caractère littéral `'0'`. La référence arrière `\g<0>` est remplacée par la sous-chaîne entière validée par l'expression rationnelle.

The expression's behaviour can be modified by specifying a *flags* value. Values can be any of the *flags* variables, combined using bitwise OR (the `|` operator).

Modifié dans la version 3.1 : ajout de l'argument optionnel *flags*.

Modifié dans la version 3.5 : Les groupes sans correspondance sont remplacés par une chaîne vide.

Modifié dans la version 3.6 : Les séquences d'échappement inconnues dans *pattern* formées par `'\'` et une lettre ASCII sont maintenant des erreurs.

Modifié dans la version 3.7 : Unknown escapes in *repl* consisting of `'\'` and an ASCII letter now are errors. Empty matches for the pattern are replaced when adjacent to a previous non-empty match.

Modifié dans la version 3.12 : Group *id* can only contain ASCII digits. In `bytes` replacement strings, group *name* can only contain bytes in the ASCII range (`b'\x00'-b'\x7f'`).

Obsolète depuis la version 3.13 : Passing *count* and *flags* as positional arguments is deprecated. In future Python versions they will be *keyword-only parameters*.

`re.`**`subn`**(*pattern*, *repl*, *string*, *count=0*, *flags=0*)

Réalise la même opération que `sub()`, mais renvoie une paire `(nouvelle_chaîne, nombre_de_substitutions_réalisées)`.

The expression's behaviour can be modified by specifying a *flags* value. Values can be any of the *flags* variables, combined using bitwise OR (the `|` operator).

`re.`**`escape`**(*pattern*)

Échappe tous les caractères spéciaux de *pattern*. Cela est utile si vous voulez valider une quelconque chaîne littérale qui pourrait contenir des métacaractères d'expressions rationnelles. Par exemple :

```
>>> print(re.escape('https://www.python.org'))
https://www\.python\.org

>>> legal_chars = string.ascii_lowercase + string.digits + "!#$%&'*+-.^_`|~:"
>>> print('[%s]+' % re.escape(legal_chars))
[abcdefghijklmnopqrstuvwxyz0123456789!\#\$%\&'\*\+\-\.\^_`\|\~:]+

>>> operators = ['+', '-', '*', '/', '**']
>>> print('|'.join(map(re.escape, sorted(operators, reverse=True))))
/|\-|\+|\*\*|\*
```

Cette fonction ne doit pas être utilisée pour la chaîne de remplacement dans `sub()` et `subn()`, seuls les antislashs devraient être échappés. Par exemple :

```
>>> digits_re = r'\d+'
>>> sample = '/usr/sbin/sendmail - 0 errors, 12 warnings'
>>> print(re.sub(digits_re, digits_re.replace('\\', r'\\'), sample))
/usr/sbin/sendmail - \d+ errors, \d+ warnings
```

Modifié dans la version 3.3 : Le caractère `'_'` n'est plus échappé.

Modifié dans la version 3.7 : Seuls les caractères qui peuvent avoir une signification spéciale dans une expression rationnelle sont échappés. De ce fait, `'!'`, `'"'`, `'%'`, `"'"`, `','`, `'/'`, `':'`, `';'`, `'<'`, `'='`, `'>'`, `'@'`, et `"`"` ne sont plus échappés.

`re.`**`purge`**()

Vide le cache d'expressions rationnelles.

Exceptions

`exception` `re.`**`PatternError`**(*msg*, *pattern=None*, *pos=None*)

Exception raised when a string passed to one of the functions here is not a valid regular expression (for example, it might contain unmatched parentheses) or when some other error occurs during compilation or matching. It

is never an error if a string contains no match for a pattern. The `PatternError` instance has the following additional attributes :

msg
Le message d'erreur non formaté.

pattern
Le motif d'expression rationnelle.

pos
L'index dans *pattern* où la compilation a échoué (peut valoir `None`).

lineno
La ligne correspondant à *pos* (peut valoir `None`).

colno
La colonne correspondant à *pos* (peut valoir `None`).

Modifié dans la version 3.5 : Ajout des attributs additionnels.

Modifié dans la version 3.13 : `PatternError` was originally named `error` ; the latter is kept as an alias for backward compatibility.

6.2.3 Objets d'expressions rationnelles

class `re.`**Pattern**

Compiled regular expression object returned by `re.compile()`.

Modifié dans la version 3.9 : `re.Pattern` supports `[]` to indicate a Unicode (str) or bytes pattern. See *Type Alias générique*.

`Pattern.`**search**(*string*[, *pos*[, *endpos*]])

Scan through *string* looking for the first location where this regular expression produces a match, and return a corresponding `Match`. Return `None` if no position in the string matches the pattern ; note that this is different from finding a zero-length match at some point in the string.

Le second paramètre *pos* (optionnel) donne l'index dans la chaîne où la recherche doit débuter ; il vaut `0` par défaut. Cela n'est pas complètement équivalent à un *slicing* sur la chaîne ; le caractère de motif `'^'` correspond au début réel de la chaîne et aux positions juste après un saut de ligne, mais pas nécessairement à l'index où la recherche commence.

Le paramètre optionnel *endpos* limite la longueur sur laquelle la chaîne sera analysée ; ce sera comme si la chaîne faisait *endpos* caractères de long, donc uniquement les caractères de *pos* à `endpos - 1` seront analysés pour trouver une correspondance. Si *endpos* est inférieur à *pos*, aucune correspondance ne sera trouvée ; dit autrement, avec *rx* une expression rationnelle compilée, `rx.search(string, 0, 50)` est équivalent à `rx.search(string[:50], 0)`.

```
>>> pattern = re.compile("d")
>>> pattern.search("dog")     # Match at index 0
<re.Match object; span=(0, 1), match='d'>
>>> pattern.search("dog", 1)  # No match; search doesn't include the "d"
```

`Pattern.`**match**(*string*[, *pos*[, *endpos*]])

If zero or more characters at the *beginning* of *string* match this regular expression, return a corresponding `Match`. Return `None` if the string does not match the pattern ; note that this is different from a zero-length match.

Les paramètres optionnels *pos* et *endpos* ont le même sens que pour la méthode `search()`.

```
>>> pattern = re.compile("o")
>>> pattern.match("dog")      # No match as "o" is not at the start of "dog".
>>> pattern.match("dog", 1)   # Match as "o" is the 2nd character of "dog".
<re.Match object; span=(1, 2), match='o'>
```

Si vous voulez une recherche n'importe où dans *string*, utilisez plutôt `search()` (voir aussi *Comparaison de search() et match()*).

`Pattern.fullmatch`(*string*[, *pos*[, *endpos*]])

If the whole *string* matches this regular expression, return a corresponding `Match`. Return `None` if the string does not match the pattern ; note that this is different from a zero-length match.

Les paramètres optionnels *pos* et *endpos* ont le même sens que pour la méthode `search()`.

```
>>> pattern = re.compile("o[gh]")
>>> pattern.fullmatch("dog")      # No match as "o" is not at the start of "dog
↪".
>>> pattern.fullmatch("ogre")     # No match as not the full string matches.
>>> pattern.fullmatch("doggie", 1, 3)   # Matches within given limits.
<re.Match object; span=(1, 3), match='og'>
```

Ajouté dans la version 3.4.

`Pattern.split`(*string*, *maxsplit=0*)

Identique à la fonction `split()`, en utilisant le motif compilé.

`Pattern.findall`(*string*[, *pos*[, *endpos*]])

Similaire à la fonction `findall()`, en utilisant le motif compilé, mais accepte aussi des paramètres *pos* et *endpos* optionnels qui limitent la région de recherche comme pour `search()`.

`Pattern.finditer`(*string*[, *pos*[, *endpos*]])

Similaire à la fonction `finditer()`, en utilisant le motif compilé, mais accepte aussi des paramètres *pos* et *endpos* optionnels qui limitent la région de recherche comme pour `search()`.

`Pattern.sub`(*repl*, *string*, *count=0*)

Identique à la fonction `sub()`, en utilisant le motif compilé.

`Pattern.subn`(*repl*, *string*, *count=0*)

Identique à la fonction `subn()`, en utilisant le motif compilé.

`Pattern.flags`

The regex matching flags. This is a combination of the flags given to `compile()`, any `(?...)` inline flags in the pattern, and implicit flags such as `UNICODE` if the pattern is a Unicode string.

`Pattern.groups`

Le nombre de groupes de capture dans le motif.

`Pattern.groupindex`

Un dictionnaire associant les noms de groupes symboliques définis par `(?P<id>)` aux groupes numérotés. Le dictionnaire est vide si aucun groupe symbolique n'est utilisé dans le motif.

`Pattern.pattern`

La chaîne de motif depuis laquelle l'objet motif a été compilé.

Modifié dans la version 3.7 : Ajout du support des fonctions `copy.copy()` et `copy.deepcopy()`. Les expressions régulières compilées sont considérées atomiques.

6.2.4 Objets de correspondance

Les objets de correspondance ont toujours une valeur booléenne `True`. Puisque `match()` et `search()` renvoient `None` quand il n'y a pas de correspondance, vous pouvez tester s'il y a eu correspondance avec une simple instruction `if` :

```
match = re.search(pattern, string)
if match:
    process(match)
```

class re.**Match**

Match object returned by successful matches and searches.

Modifié dans la version 3.9 : *re.Match* supports [] to indicate a Unicode (str) or bytes match. See *Type Alias générique*.

Match.**expand**(*template*)

Return the string obtained by doing backslash substitution on the template string *template*, as done by the *sub()* method. Escapes such as \n are converted to the appropriate characters, and numeric backreferences (\1, \2) and named backreferences (\g<1>, \g<name>) are replaced by the contents of the corresponding group. The backreference \g<0> will be replaced by the entire match.

Modifié dans la version 3.5 : Les groupes sans correspondance sont remplacés par une chaîne vide.

Match.**group**([*group1*, ...])

Renvoie un ou plus sous-groupes de la correspondance. Si un seul argument est donné, le résultat est une chaîne simple ; s'il y a plusieurs arguments, le résultat est un *n*-uplet comprenant un élément par argument. Sans arguments, *group1* vaut par défaut zéro (la correspondance entière est renvoyée). Si un argument *groupN* vaut zéro, l'élément associé sera la chaîne de correspondance entière ; s'il est dans l'intervalle fermé [1..99], c'est la correspondance avec le groupe de parenthèses associé. Si un numéro de groupe est négatif ou supérieur au nombre de groupes définis dans le motif, une exception indexError est levée. Si un groupe est contenu dans une partie du motif qui n'a aucune correspondance, l'élément associé sera None. Si un groupe est contenu dans une partie du motif qui a plusieurs correspondances, seule la dernière correspondance est renvoyée.

```
>>> m = re.match(r"(\w+) (\w+)", "Isaac Newton, physicist")
>>> m.group(0)       # The entire match
'Isaac Newton'
>>> m.group(1)       # The first parenthesized subgroup.
'Isaac'
>>> m.group(2)       # The second parenthesized subgroup.
'Newton'
>>> m.group(1, 2)    # Multiple arguments give us a tuple.
('Isaac', 'Newton')
```

Si l'expression rationnelle utilise la syntaxe (?P<name>...), les arguments *groupN* peuvent alors aussi être des chaînes identifiant les groupes par leurs noms. Si une chaîne donnée en argument n'est pas utilisée comme nom de groupe dans le motif, une exception *IndexError* est levée.

Un exemple modérément compliqué :

```
>>> m = re.match(r"(?P<first_name>\w+) (?P<last_name>\w+)", "Malcolm Reynolds")
>>> m.group('first_name')
'Malcolm'
>>> m.group('last_name')
'Reynolds'
```

Les groupes nommés peuvent aussi être référencés par leur index :

```
>>> m.group(1)
'Malcolm'
>>> m.group(2)
'Reynolds'
```

Si un groupe a plusieurs correspondances, seule la dernière est accessible :

```
>>> m = re.match(r"(..)+", "a1b2c3")  # Matches 3 times.
>>> m.group(1)                        # Returns only the last match.
'c3'
```

Match.**__getitem__**(*g*)

Cela est identique à `m.group(g)`. Cela permet un accès plus facile à un groupe individuel depuis une correspondance :

```
>>> m = re.match(r"(\w+) (\w+)", "Isaac Newton, physicist")
>>> m[0]       # The entire match
'Isaac Newton'
>>> m[1]       # The first parenthesized subgroup.
'Isaac'
>>> m[2]       # The second parenthesized subgroup.
'Newton'
```

Named groups are supported as well :

```
>>> m = re.match(r"(?P<first_name>\w+) (?P<last_name>\w+)", "Isaac Newton")
>>> m['first_name']
'Isaac'
>>> m['last_name']
'Newton'
```

Ajouté dans la version 3.6.

`Match.groups`(*default=None*)

Renvoie un *n*-uplet contenant tous les sous-groupes de la correspondance, de 1 jusqu'au nombre de groupes dans le motif. L'argument *default* est utilisé pour les groupes sans correspondance ; il vaut `None` par défaut.

Par exemple :

```
>>> m = re.match(r"(\d+)\.(\d+)", "24.1632")
>>> m.groups()
('24', '1632')
```

Si on rend la partie décimale et tout ce qui la suit optionnels, tous les groupes ne figureront pas dans la correspondance. Ces groupes sans correspondance vaudront `None` sauf si une autre valeur est donnée à l'argument *default* :

```
>>> m = re.match(r"(\d+)\.?(\d+)?", "24")
>>> m.groups()      # Second group defaults to None.
('24', None)
>>> m.groups('0')   # Now, the second group defaults to '0'.
('24', '0')
```

`Match.groupdict`(*default=None*)

Renvoie un dictionnaire contenant tous les sous-groupes *nommés* de la correspondance, accessibles par leurs noms. L'argument *default* est utilisé pour les groupes qui ne figurent pas dans la correspondance ; il vaut `None` par défaut. Par exemple :

```
>>> m = re.match(r"(?P<first_name>\w+) (?P<last_name>\w+)", "Malcolm Reynolds")
>>> m.groupdict()
{'first_name': 'Malcolm', 'last_name': 'Reynolds'}
```

`Match.start`([*group*])

`Match.end`([*group*])

Renvoie les indices de début et de fin de la sous-chaîne correspondant au groupe *group* ; *group* vaut par défaut zéro (pour récupérer les indices de la correspondance complète). Renvoie `-1` si *group* existe mais ne figure pas dans la correspondance. Pour un objet de correspondance *m*, et un groupe *g* qui y figure, la sous-chaîne correspondant au groupe *g* (équivalente à `m.group(g)`) est

```
m.string[m.start(g):m.end(g)]
```

Notez que `m.start(group)` sera égal à `m.end(group)` si *group* correspond à une chaîne vide. Par exemple, après `m = re.search('b(c?)', 'cba')`, `m.start(0)` vaut 1, `m.end(0)` vaut 2, `m.start(1)` et `m.end(1)` valent tous deux 2, et `m.start(2)` lève une exception `IndexError`.

Un exemple qui supprimera *remove_this* d'une adresse mail :

```
>>> email = "tony@tiremove_thisger.net"
>>> m = re.search("remove_this", email)
>>> email[:m.start()] + email[m.end():]
'tony@tiger.net'
```

`Match.`**`span`**`(`[*group*]`)`

Pour un objet de correspondance *m*, renvoie la paire `(m.start(group), m.end(group))`. Notez que si *group* ne figure pas dans la correspondance, `(-1, -1)` est renvoyé. *group* vaut par défaut zéro, pour la correspondance entière.

`Match.`**`pos`**

La valeur de *pos* qui a été passée à la méthode `search()` ou `match()` d'un *objet expression rationnelle*. C'est l'index dans la chaîne à partir duquel le moteur d'expressions rationnelles recherche une correspondance.

`Match.`**`endpos`**

La valeur de *endpos* qui a été passée à la méthode `search()` ou `match()` d'un *objet expression rationnelle*. C'est l'index dans la chaîne que le moteur d'expressions rationnelles ne dépassera pas.

`Match.`**`lastindex`**

L'index entier du dernier groupe de capture validé, ou `None` si aucun groupe ne correspondait. Par exemple, les expressions `(a)b`, `((a)(b))` et `((ab))` auront un `lastindex == 1` si appliquées à la chaîne `'ab'`, alors que l'expression `(a)(b)` aura un `lastindex == 2` si appliquée à la même chaîne.

`Match.`**`lastgroup`**

Le nom du dernier groupe capturant validé, ou `None` si le groupe n'a pas de nom, ou si aucun groupe ne correspondait.

`Match.`**`re`**

L'expression rationnelle dont la méthode `match()` ou `search()` a produit cet objet de correspondance.

`Match.`**`string`**

La chaîne passée à `match()` ou `search()`.

Modifié dans la version 3.7 : Ajout du support des fonctions `copy.copy()` et `copy.deepcopy()`. Les objets correspondants sont considérés atomiques.

6.2.5 Exemples d'expressions rationnelles

Rechercher une paire

Dans cet exemple, nous nous aidons de la fonction suivante pour afficher de manière plus jolie les objets qui correspondent :

```
def displaymatch(match):
    if match is None:
        return None
    return '<Match: %r, groups=%r>' % (match.group(), match.groups())
```

Supposez que vous écriviez un jeu de poker où la main d'un joueur est représentée par une chaîne de 5 caractères avec chaque caractère représentant une carte, « a » pour l'as, « k » pour le roi (*king*), « q » pour la reine (*queen*), « j » pour le valet (*jack*), « t » pour 10 (*ten*), et les caractères de « 2 » à « 9 » représentant les cartes avec ces valeurs.

Pour vérifier qu'une chaîne donnée est une main valide, on pourrait faire comme suit :

```
>>> valid = re.compile(r"^[a2-9tjqk]{5}$")
>>> displaymatch(valid.match("akt5q"))  # Valid.
"<Match: 'akt5q', groups=()>"
>>> displaymatch(valid.match("akt5e"))  # Invalid.
>>> displaymatch(valid.match("akt"))    # Invalid.
>>> displaymatch(valid.match("727ak"))  # Valid.
"<Match: '727ak', groups=()>"
```

La dernière main, `"727ak"`, contenait une paire, deux cartes de la même valeur. Pour valider cela avec une expression rationnelle, on pourrait utiliser des références arrière comme :

```
>>> pair = re.compile(r".*(.).*\1")
>>> displaymatch(pair.match("717ak"))     # Pair of 7s.
"<Match: '717', groups=('7',)>"
>>> displaymatch(pair.match("718ak"))     # No pairs.
>>> displaymatch(pair.match("354aa"))     # Pair of aces.
"<Match: '354aa', groups=('a',)>"
```

Pour trouver de quelle carte est composée la paire, on pourrait utiliser la méthode `group()` de l'objet de correspondance de la manière suivante :

```
>>> pair = re.compile(r".*(.).*\1")
>>> pair.match("717ak").group(1)
'7'

# Error because re.match() returns None, which doesn't have a group() method:
>>> pair.match("718ak").group(1)
Traceback (most recent call last):
  File "<pyshell#23>", line 1, in <module>
    re.match(r".*(.).*\1", "718ak").group(1)
AttributeError: 'NoneType' object has no attribute 'group'

>>> pair.match("354aa").group(1)
'a'
```

Simuler *scanf()*

Python does not currently have an equivalent to `scanf()`. Regular expressions are generally more powerful, though also more verbose, than `scanf()` format strings. The table below offers some more-or-less equivalent mappings between `scanf()` format tokens and regular expressions.

`scanf()` Token	Expression rationnelle
`%c`	`.`
`%5c`	`.{5}`
`%d`	`[-+]?\d+`
`%e, %E, %f, %g`	`[-+]?(\d+(\.\d*)?|\.\d+)([eE][-+]?\d+)?`
`%i`	`[-+]?(0[xX][\dA-Fa-f]+|0[0-7]*|\d+)`
`%o`	`[-+]?[0-7]+`
`%s`	`\S+`
`%u`	`\d+`
`%x, %X`	`[-+]?(0[xX])?[\dA-Fa-f]+`

Pour extraire le nom de fichier et les nombres depuis une chaîne comme

```
/usr/sbin/sendmail - 0 errors, 4 warnings
```

you would use a `scanf()` format like

```
%s - %d errors, %d warnings
```

L'expression rationnelle équivalente serait

```
(\S+) - (\d+) errors, (\d+) warnings
```

Comparaison de *search()* et *match()*

Python offre différentes opérations primitives basées sur des expressions régulières :

- `re.match()` cherche une correspondance uniquement au début de la chaîne de caractères
- `re.search()` cherche une correspondance n'importe où dans la chaîne de caractères (ce que fait Perl par défaut)
- `re.fullmatch()` cherche une correspondance avec l'intégralité de la chaîne de caractères.

Par exemple :

```
>>> re.match("c", "abcdef")     # No match
>>> re.search("c", "abcdef")    # Match
<re.Match object; span=(2, 3), match='c'>
>>> re.fullmatch("p.*n", "python") # Match
<re.Match object; span=(0, 6), match='python'>
>>> re.fullmatch("r.*n", "python") # No match
```

Les expressions rationnelles commençant par `'^'` peuvent être utilisées avec `search()` pour restreindre la recherche au début de la chaîne :

```
>>> re.match("c", "abcdef")     # No match
>>> re.search("^c", "abcdef")   # No match
>>> re.search("^a", "abcdef")   # Match
<re.Match object; span=(0, 1), match='a'>
```

Notez cependant qu'en mode `MULTILINE`, `match()` ne recherche qu'au début de la chaîne, alors que `search()` avec une expression rationnelle commençant par `'^'` recherchera au début de chaque ligne.

```
>>> re.match("X", "A\nB\nX", re.MULTILINE)  # No match
>>> re.search("^X", "A\nB\nX", re.MULTILINE)  # Match
<re.Match object; span=(4, 5), match='X'>
```

Construire un répertoire téléphonique

`split()` découpe une chaîne en une liste délimitée par le motif donné. La méthode est inestimable pour convertir des données textuelles vers des structures de données qui peuvent être lues et modifiées par Python comme démontré dans l'exemple suivant qui crée un répertoire téléphonique.

Tout d'abord, voici l'entrée. Elle provient normalement d'un fichier, nous utilisons ici une chaîne à guillemets triples

```
>>> text = """Ross McFluff: 834.345.1254 155 Elm Street
...
... Ronald Heathmore: 892.345.3428 436 Finley Avenue
... Frank Burger: 925.541.7625 662 South Dogwood Way
...
...
... Heather Albrecht: 548.326.4584 919 Park Place"""
```

Les entrées sont séparées par un saut de ligne ou plus. Nous convertissons maintenant la chaîne en une liste où chaque ligne non vide aura sa propre entrée :

```
>>> entries = re.split("\n+", text)
>>> entries
```

(suite sur la page suivante)

(suite de la page précédente)

```
['Ross McFluff: 834.345.1254 155 Elm Street',
'Ronald Heathmore: 892.345.3428 436 Finley Avenue',
'Frank Burger: 925.541.7625 662 South Dogwood Way',
'Heather Albrecht: 548.326.4584 919 Park Place']
```

Finalement, on sépare chaque entrée en une liste avec prénom, nom, numéro de téléphone et adresse. Nous utilisons le paramètre `maxsplit` de *split()* parce que l'adresse contient des espaces, qui sont notre motif de séparation :

```
>>> [re.split(":? ", entry, maxsplit=3) for entry in entries]
[['Ross', 'McFluff', '834.345.1254', '155 Elm Street'],
['Ronald', 'Heathmore', '892.345.3428', '436 Finley Avenue'],
['Frank', 'Burger', '925.541.7625', '662 South Dogwood Way'],
['Heather', 'Albrecht', '548.326.4584', '919 Park Place']]
```

Le motif `:?` trouve les deux points derrière le nom de famille, pour qu'ils n'apparaissent pas dans la liste résultante. Avec un `maxsplit` de `4`, nous pourrions séparer le numéro du nom de la rue :

```
>>> [re.split(":? ", entry, maxsplit=4) for entry in entries]
[['Ross', 'McFluff', '834.345.1254', '155', 'Elm Street'],
['Ronald', 'Heathmore', '892.345.3428', '436', 'Finley Avenue'],
['Frank', 'Burger', '925.541.7625', '662', 'South Dogwood Way'],
['Heather', 'Albrecht', '548.326.4584', '919', 'Park Place']]
```

Mélanger les lettres des mots

sub() remplace toutes les occurrences d'un motif par une chaîne ou le résultat d'une fonction. Cet exemple le montre, en utilisant *sub()* avec une fonction qui mélange aléatoirement les caractères de chaque mot dans une phrase (à l'exception des premiers et derniers caractères) :

```
>>> def repl(m):
...     inner_word = list(m.group(2))
...     random.shuffle(inner_word)
...     return m.group(1) + "".join(inner_word) + m.group(3)
...
>>> text = "Professor Abdolmalek, please report your absences promptly."
>>> re.sub(r"(\w)(\w+)(\w)", repl, text)
'Poefsrosr Aealmlobdk, pslaee reorpt your abnseces plmrptoy.'
>>> re.sub(r"(\w)(\w+)(\w)", repl, text)
'Pofsroser Aodlambelk, plasee reoprt yuor asnebces potlmrpy.'
```

Trouver tous les adverbes

findall() trouve *toutes* les occurrences d'un motif, pas juste la première comme le fait *search()*. Par exemple, si un écrivain voulait trouver tous les adverbes dans un texte, il devrait utiliser *findall()* de la manière suivante :

```
>>> text = "He was carefully disguised but captured quickly by police."
>>> re.findall(r"\w+ly\b", text)
['carefully', 'quickly']
```

Trouver tous les adverbes et leurs positions

If one wants more information about all matches of a pattern than the matched text, *finditer()* is useful as it provides *Match* objects instead of strings. Continuing with the previous example, if a writer wanted to find all of the adverbs *and their positions* in some text, they would use *finditer()* in the following manner :

```
>>> text = "He was carefully disguised but captured quickly by police."
>>> for m in re.finditer(r"\w+ly\b", text):
...     print('%02d-%02d: %s' % (m.start(), m.end(), m.group(0)))
07-16: carefully
40-47: quickly
```

Notation brute de chaînes

La notation brute de chaînes (`r"text"`) garde saines les expressions rationnelles. Sans elle, chaque *backslash* (`'\'`) dans une expression rationnelle devrait être préfixé d'un autre *backslash* pour l'échapper. Par exemple, les deux lignes de code suivantes sont fonctionnellement identiques :

```
>>> re.match(r"\W(.)\1\W", " ff ")
<re.Match object; span=(0, 4), match=' ff '>
>>> re.match("\\W(.)\\1\\W", " ff ")
<re.Match object; span=(0, 4), match=' ff '>
```

Pour rechercher un *backslash* littéral, il faut l'échapper dans l'expression rationnelle. Avec la notation brute, cela signifie `r"\\"`. Sans elle, il faudrait utiliser `"\\\\"`, faisant que les deux lignes de code suivantes sont fonctionnellement identiques :

```
>>> re.match(r"\\", r"\\")
<re.Match object; span=(0, 1), match='\\'>
>>> re.match("\\\\", r"\\")
<re.Match object; span=(0, 1), match='\\'>
```

Écrire un analyseur lexical

Un analyseur lexical ou scanner analyse une chaîne pour catégoriser les groupes de caractères. C'est une première étape utile dans l'écriture d'un compilateur ou d'un interpréteur.

Les catégories de texte sont spécifiées par des expressions rationnelles. La technique est de les combiner dans une unique expression rationnelle maîtresse, et de boucler sur les correspondances successives :

```
from typing import NamedTuple
import re

class Token(NamedTuple):
    type: str
    value: str
    line: int
    column: int

def tokenize(code):
    keywords = {'IF', 'THEN', 'ENDIF', 'FOR', 'NEXT', 'GOSUB', 'RETURN'}
    token_specification = [
        ('NUMBER',   r'\d+(\.\d*)?'),  # Integer or decimal number
        ('ASSIGN',   r':='),           # Assignment operator
        ('END',      r';'),            # Statement terminator
        ('ID',       r'[A-Za-z]+'),    # Identifiers
        ('OP',       r'[+\-*/]'),      # Arithmetic operators
        ('NEWLINE',  r'\n'),           # Line endings
        ('SKIP',     r'[ \t]+'),       # Skip over spaces and tabs
        ('MISMATCH', r'.'),            # Any other character
    ]
    tok_regex = '|'.join('(?P<%s>%s)' % pair for pair in token_specification)
    line_num = 1
```

(suite sur la page suivante)

(suite de la page précédente)

```
    line_start = 0
    for mo in re.finditer(tok_regex, code):
        kind = mo.lastgroup
        value = mo.group()
        column = mo.start() - line_start
        if kind == 'NUMBER':
            value = float(value) if '.' in value else int(value)
        elif kind == 'ID' and value in keywords:
            kind = value
        elif kind == 'NEWLINE':
            line_start = mo.end()
            line_num += 1
            continue
        elif kind == 'SKIP':
            continue
        elif kind == 'MISMATCH':
            raise RuntimeError(f'{value!r} unexpected on line {line_num}')
        yield Token(kind, value, line_num, column)

statements = '''
    IF quantity THEN
        total := total + price * quantity;
        tax := price * 0.05;
    ENDIF;
'''

for token in tokenize(statements):
    print(token)
```

L'analyseur produit la sortie suivante :

```
Token(type='IF', value='IF', line=2, column=4)
Token(type='ID', value='quantity', line=2, column=7)
Token(type='THEN', value='THEN', line=2, column=16)
Token(type='ID', value='total', line=3, column=8)
Token(type='ASSIGN', value=':=', line=3, column=14)
Token(type='ID', value='total', line=3, column=17)
Token(type='OP', value='+', line=3, column=23)
Token(type='ID', value='price', line=3, column=25)
Token(type='OP', value='*', line=3, column=31)
Token(type='ID', value='quantity', line=3, column=33)
Token(type='END', value=';', line=3, column=41)
Token(type='ID', value='tax', line=4, column=8)
Token(type='ASSIGN', value=':=', line=4, column=12)
Token(type='ID', value='price', line=4, column=15)
Token(type='OP', value='*', line=4, column=21)
Token(type='NUMBER', value=0.05, line=4, column=23)
Token(type='END', value=';', line=4, column=27)
Token(type='ENDIF', value='ENDIF', line=5, column=4)
Token(type='END', value=';', line=5, column=9)
```

6.3 `difflib` --- Helpers for computing deltas

Code source : Lib/difflib.py

This module provides classes and functions for comparing sequences. It can be used for example, for comparing files, and can produce information about file differences in various formats, including HTML and context and unified diffs. For comparing directories and files, see also, the `filecmp` module.

class difflib.**SequenceMatcher**

C'est une classe flexible permettant de comparer des séquences deux à deux de n'importe quel type, tant que les éléments des séquences sont *hachables*. L'algorithme de base est antérieur, et un peu plus sophistiqué, à un algorithme publié à la fin des années 1980 par Ratcliff et Obershelp sous le nom hyperbolique de *gestalt pattern matching*. L'idée est de trouver la plus longue sous-séquence d'appariement contiguë qui ne contient pas d'éléments « indésirables » ; ces éléments « indésirables » sont ceux qui sont inintéressants dans un certain sens, comme les lignes blanches ou les espaces. (Le traitement des éléments indésirables est une extension de l'algorithme de Ratcliff et Obershelp). La même idée est ensuite appliquée récursivement aux morceaux des séquences à gauche et à droite de la sous-séquence correspondante. Cela ne donne pas des séquences de montage minimales, mais tend à donner des correspondances qui « semblent correctes » pour les gens.

Compléxité temporelle : l'algorithme de base de Ratcliff-Obershelp est de complexité cubique dans le pire cas et de complexité quadratique dans le cas attendu. `SequenceMatcher` est de complexité quadratique pour le pire cas et son comportement dans le cas attendu dépend de façon complexe du nombre d'éléments que les séquences ont en commun ; le temps dans le meilleur cas est linéaire.

Heuristique automatique des indésirables : `SequenceMatcher` utilise une heuristique qui traite automatiquement certains éléments de la séquence comme indésirables. L'heuristique compte combien de fois chaque élément individuel apparaît dans la séquence. Si les doublons d'un élément (après le premier) représentent plus de 1 % de la séquence et que la séquence compte au moins 200 éléments, cet élément est marqué comme « populaire » et est traité comme indésirable aux fins de la comparaison des séquences. Cette heuristique peut être désactivée en réglant l'argument `autojunk` sur `False` lors de la création de la classe `SequenceMatcher`.

Modifié dans la version 3.2 : Added the *autojunk* parameter.

class difflib.**Differ**

Il s'agit d'une classe permettant de comparer des séquences de lignes de texte et de produire des différences ou deltas humainement lisibles. *Differ* utilise `SequenceMatcher` à la fois pour comparer des séquences de lignes, et pour comparer des séquences de caractères dans des lignes similaires (quasi-correspondantes).

Chaque ligne d'un delta `Differ` commence par un code de deux lettres :

Code	Signification
`'- '`	ligne n'appartenant qu'à la séquence 1
`'+ '`	ligne n'appartenant qu'à la séquence 2
`' '`	ligne commune aux deux séquences
`'? '`	ligne non présente dans l'une ou l'autre des séquences d'entrée

Lines beginning with '`?`' attempt to guide the eye to intraline differences, and were not present in either input sequence. These lines can be confusing if the sequences contain whitespace characters, such as spaces, tabs or line breaks.

class difflib.**HtmlDiff**

Cette classe peut être utilisée pour créer un tableau HTML (ou un fichier HTML complet contenant le tableau) montrant une comparaison côte à côte, ligne par ligne, du texte avec les changements inter-lignes et intralignes. Le tableau peut être généré en mode de différence complet ou contextuel.

Le constructeur pour cette classe est :

__init__(*tabsize=8*, *wrapcolumn=None*, *linejunk=None*, *charjunk=IS_CHARACTER_JUNK*)

Initialise l'instance de `HtmlDiff`.

tabsize est argument nommé optionnel pour spécifier l'espacement des tabulations et sa valeur par défaut est `8`.

wrapcolumn est un argument nommé optionnel pour spécifier le numéro de la colonne où les lignes sont coupées pour être ré-agencées, la valeur par défaut est `None` lorsque les lignes ne sont pas ré-agencées.

linejunk et *charjunk* sont des arguments nommés optionnels passés dans `ndiff()` (utilisés par `HtmlDiff` pour générer les différences HTML côte à côte). Voir la documentation de `ndiff()` pour les valeurs par défaut des arguments et les descriptions.

Les méthodes suivantes sont publiques :

make_file(*fromlines*, *tolines*, *fromdesc=''*, *todesc=''*, *context=False*, *numlines=5*, **, charset='utf-8'*)

Compare *fromlines* et *tolines* (listes de chaînes de caractères) et renvoie une chaîne de caractères qui est un fichier HTML complet contenant un tableau montrant les différences ligne par ligne avec les changements inter-lignes et intralignes mis en évidence.

fromdesc et *todesc* sont des arguments nommés optionnels pour spécifier les chaînes d'en-tête des colonnes *from/to* du fichier (les deux sont des chaînes vides par défaut).

context et *numlines* sont tous deux des arguments nommés facultatifs. Mettre *context* à `True` lorsque les différences contextuelles doivent être affichées, sinon la valeur par défaut est `False` pour afficher les fichiers complets. Les *numlines* ont pour valeur par défaut `5`. Lorsque *context* est `True`, *numlines* contrôle le nombre de lignes de contexte qui entourent les différences mise en évidence. Lorsque *context* est `False`, *numlines* contrôle le nombre de lignes qui sont affichées avant un surlignage de différence lors de l'utilisation des hyperliens « suivants » (un réglage à zéro ferait en sorte que les hyperliens « suivants » placeraient le surlignage de différence suivant en haut du navigateur sans aucun contexte introductif).

> **Note**
>
> *fromdesc* et *todesc* sont interprétés comme du HTML non échappé et doivent être correctement échappés lors de la réception de données provenant de sources non fiables.

Modifié dans la version 3.5 : l'argument nommé *charset* a été ajouté. Le jeu de caractères par défaut du document HTML est passé de `'ISO-8859-1'` à `'utf-8'`.

make_table(*fromlines*, *tolines*, *fromdesc=''*, *todesc=''*, *context=False*, *numlines=5*)

Compare *fromlines* et *tolines* (listes de chaînes) et renvoie une chaîne qui est un tableau HTML complet montrant les différences ligne par ligne avec les changements inter-lignes et intralignes mis en évidence.

Les arguments pour cette méthode sont les mêmes que ceux de la méthode `make_file()`.

difflib.**context_diff**(*a*, *b*, *fromfile=''*, *tofile=''*, *fromfiledate=''*, *tofiledate=''*, *n=3*, *lineterm='\n'*)

Compare *a* et *b* (listes de chaînes de caractères) ; renvoie un delta (un *generateur* générant les lignes delta) dans un format de différence de contexte.

Context diffs are a compact way of showing just the lines that have changed plus a few lines of context. The changes are shown in a before/after style. The number of context lines is set by *n* which defaults to three.

Par défaut, les lignes de contrôle de la différence (celles avec `***` ou `---`) sont créées avec un saut de ligne à la fin. Ceci est utile pour que les entrées créées à partir de `io.IOBase.readlines()` résultent en des différences qui peuvent être utilisées avec `io.IOBase.writelines()` puisque les entrées et les sorties ont des nouvelles lignes de fin.

Pour les entrées qui n'ont pas de retour à la ligne, mettre l'argument *lineterm* à `""` afin que la sortie soit uniformément sans retour à la ligne.

Le format de contexte de différence comporte normalement un en-tête pour les noms de fichiers et les heures de modification. Tout ou partie de ces éléments peuvent être spécifiés en utilisant les chaînes de caractères *fromfile*, *tofile*, *fromfiledate* et *tofiledate*. Les heures de modification sont normalement exprimées dans le format ISO 8601. Si elles ne sont pas spécifiées, les chaînes de caractères sont par défaut vierges.

```
>>> import sys
>>> from difflib import *
>>> s1 = ['bacon\n', 'eggs\n', 'ham\n', 'guido\n']
>>> s2 = ['python\n', 'eggy\n', 'hamster\n', 'guido\n']
>>> sys.stdout.writelines(context_diff(s1, s2, fromfile='before.py',
```

(suite sur la page suivante)

(suite de la page précédente)

```
...                             tofile='after.py'))
*** before.py
--- after.py
***************
*** 1,4 ****
! bacon
! eggs
! ham
  guido
--- 1,4 ----
! python
! eggy
! hamster
  guido
```

Voir *une interface de ligne de commandes pour difflib* pour un exemple plus détaillé.

difflib.**get_close_matches**(*word*, *possibilities*, *n=3*, *cutoff=0.6*)

Return a list of the best "good enough" matches. *word* is a sequence for which close matches are desired (typically a string), and *possibilities* is a list of sequences against which to match *word* (typically a list of strings).

Optional argument *n* (default `3`) is the maximum number of close matches to return ; *n* must be greater than `0`.

Optional argument *cutoff* (default `0.6`) is a float in the range [0, 1]. Possibilities that don't score at least that similar to *word* are ignored.

The best (no more than *n*) matches among the possibilities are returned in a list, sorted by similarity score, most similar first.

```
>>> get_close_matches('appel', ['ape', 'apple', 'peach', 'puppy'])
['apple', 'ape']
>>> import keyword
>>> get_close_matches('wheel', keyword.kwlist)
['while']
>>> get_close_matches('pineapple', keyword.kwlist)
[]
>>> get_close_matches('accept', keyword.kwlist)
['except']
```

difflib.**ndiff**(*a*, *b*, *linejunk=None*, *charjunk=IS_CHARACTER_JUNK*)

Compare *a* and *b* (lists of strings) ; return a `Differ`-style delta (a *generator* generating the delta lines).

Optional keyword parameters *linejunk* and *charjunk* are filtering functions (or `None`) :

linejunk : A function that accepts a single string argument, and returns true if the string is junk, or false if not. The default is `None`. There is also a module-level function `IS_LINE_JUNK()`, which filters out lines without visible characters, except for at most one pound character (`'#'`) -- however the underlying `SequenceMatcher` class does a dynamic analysis of which lines are so frequent as to constitute noise, and this usually works better than using this function.

charjunk : A function that accepts a character (a string of length 1), and returns if the character is junk, or false if not. The default is module-level function `IS_CHARACTER_JUNK()`, which filters out whitespace characters (a blank or tab ; it's a bad idea to include newline in this !).

```
>>> diff = ndiff('one\ntwo\nthree\n'.splitlines(keepends=True),
...              'ore\ntree\nemu\n'.splitlines(keepends=True))
>>> print(''.join(diff), end="")
- one
?  ^
```

(suite sur la page suivante)

(suite de la page précédente)

```
+ ore
?  ^
- two
- three
?  -
+ tree
+ emu
```

difflib.**restore**(*sequence*, *which*)

Return one of the two sequences that generated a delta.

Given a *sequence* produced by `Differ.compare()` or `ndiff()`, extract lines originating from file 1 or 2 (parameter *which*), stripping off line prefixes.

Exemple :

```
>>> diff = ndiff('one\ntwo\nthree\n'.splitlines(keepends=True),
...              'ore\ntree\nemu\n'.splitlines(keepends=True))
>>> diff = list(diff) # materialize the generated delta into a list
>>> print(''.join(restore(diff, 1)), end="")
one
two
three
>>> print(''.join(restore(diff, 2)), end="")
ore
tree
emu
```

difflib.**unified_diff**(*a*, *b*, *fromfile=''*, *tofile=''*, *fromfiledate=''*, *tofiledate=''*, *n=3*, *lineterm='\n'*)

Compare *a* et *b* (lists of strings) ; return a delta (a *generator* generating the delta lines) in unified diff format.

Unified diffs are a compact way of showing just the lines that have changed plus a few lines of context. The changes are shown in an inline style (instead of separate before/after blocks). The number of context lines is set by *n* which defaults to three.

By default, the diff control lines (those with `---`, `+++`, or `@@`) are created with a trailing newline. This is helpful so that inputs created from `io.IOBase.readlines()` result in diffs that are suitable for use with `io.IOBase.writelines()` since both the inputs and outputs have trailing newlines.

Pour les entrées qui n'ont pas de retour à la ligne, mettre l'argument *lineterm* à `""` afin que la sortie soit uniformément sans retour à la ligne.

The unified diff format normally has a header for filenames and modification times. Any or all of these may be specified using strings for *fromfile*, *tofile*, *fromfiledate*, and *tofiledate*. The modification times are normally expressed in the ISO 8601 format. If not specified, the strings default to blanks.

```
>>> s1 = ['bacon\n', 'eggs\n', 'ham\n', 'guido\n']
>>> s2 = ['python\n', 'eggy\n', 'hamster\n', 'guido\n']
>>> sys.stdout.writelines(unified_diff(s1, s2, fromfile='before.py', tofile=
↪'after.py'))
--- before.py
+++ after.py
@@ -1,4 +1,4 @@
-bacon
-eggs
-ham
+python
+eggy
+hamster
```

(suite sur la page suivante)

(suite de la page précédente)

```
guido
```

Voir *une interface de ligne de commandes pour difflib* pour un exemple plus détaillé.

difflib.**diff_bytes**(*dfunc*, *a*, *b*, *fromfile=b''*, *tofile=b''*, *fromfiledate=b''*, *tofiledate=b''*, *n=3*, *lineterm=b'\n'*)

Compare *a* and *b* (lists of bytes objects) using *dfunc* ; yield a sequence of delta lines (also bytes) in the format returned by *dfunc*. *dfunc* must be a callable, typically either `unified_diff()` or `context_diff()`.

Allows you to compare data with unknown or inconsistent encoding. All inputs except *n* must be bytes objects, not str. Works by losslessly converting all inputs (except *n*) to str, and calling `dfunc(a, b, fromfile, tofile, fromfiledate, tofiledate, n, lineterm)`. The output of *dfunc* is then converted back to bytes, so the delta lines that you receive have the same unknown/inconsistent encodings as *a* and *b*.

Ajouté dans la version 3.5.

difflib.**IS_LINE_JUNK**(*line*)

Return `True` for ignorable lines. The line *line* is ignorable if *line* is blank or contains a single `'#'`, otherwise it is not ignorable. Used as a default for parameter *linejunk* in `ndiff()` in older versions.

difflib.**IS_CHARACTER_JUNK**(*ch*)

Return `True` for ignorable characters. The character *ch* is ignorable if *ch* is a space or tab, otherwise it is not ignorable. Used as a default for parameter *charjunk* in `ndiff()`.

Voir aussi

Pattern Matching : The Gestalt Approach
Discussion of a similar algorithm by John W. Ratcliff and D. E. Metzener. This was published in Dr. Dobb's Journal in July, 1988.

6.3.1 SequenceMatcher Objects

The `SequenceMatcher` class has this constructor :

class difflib.**SequenceMatcher**(*isjunk=None*, *a=''*, *b=''*, *autojunk=True*)

Optional argument *isjunk* must be `None` (the default) or a one-argument function that takes a sequence element and returns true if and only if the element is "junk" and should be ignored. Passing `None` for *isjunk* is equivalent to passing `lambda x: False` ; in other words, no elements are ignored. For example, pass :

```
lambda x: x in " \t"
```

if you're comparing lines as sequences of characters, and don't want to synch up on blanks or hard tabs.

The optional arguments *a* and *b* are sequences to be compared ; both default to empty strings. The elements of both sequences must be *hashable*.

The optional argument *autojunk* can be used to disable the automatic junk heuristic.

Modifié dans la version 3.2 : Added the *autojunk* parameter.

SequenceMatcher objects get three data attributes : *bjunk* is the set of elements of *b* for which *isjunk* is `True` ; *bpopular* is the set of non-junk elements considered popular by the heuristic (if it is not disabled) ; *b2j* is a dict mapping the remaining elements of *b* to a list of positions where they occur. All three are reset whenever *b* is reset with `set_seqs()` or `set_seq2()`.

Ajouté dans la version 3.2 : The *bjunk* and *bpopular* attributes.

`SequenceMatcher` objects have the following methods :

set_seqs(*a*, *b*)

Set the two sequences to be compared.

SequenceMatcher computes and caches detailed information about the second sequence, so if you want to compare one sequence against many sequences, use *set_seq2()* to set the commonly used sequence once and call *set_seq1()* repeatedly, once for each of the other sequences.

set_seq1(*a*)

Set the first sequence to be compared. The second sequence to be compared is not changed.

set_seq2(*b*)

Set the second sequence to be compared. The first sequence to be compared is not changed.

find_longest_match(*alo=0*, *ahi=None*, *blo=0*, *bhi=None*)

Find longest matching block in `a[alo:ahi]` and `b[blo:bhi]`.

If *isjunk* was omitted or `None`, *find_longest_match()* returns `(i, j, k)` such that `a[i:i+k]` is equal to `b[j:j+k]`, where `alo <= i <= i+k <= ahi` and `blo <= j <= j+k <= bhi`. For all `(i', j', k')` meeting those conditions, the additional conditions `k >= k'`, `i <= i'`, and if `i == i'`, `j <= j'` are also met. In other words, of all maximal matching blocks, return one that starts earliest in *a*, and of all those maximal matching blocks that start earliest in *a*, return the one that starts earliest in *b*.

```
>>> s = SequenceMatcher(None, " abcd", "abcd abcd")
>>> s.find_longest_match(0, 5, 0, 9)
Match(a=0, b=4, size=5)
```

If *isjunk* was provided, first the longest matching block is determined as above, but with the additional restriction that no junk element appears in the block. Then that block is extended as far as possible by matching (only) junk elements on both sides. So the resulting block never matches on junk except as identical junk happens to be adjacent to an interesting match.

Here's the same example as before, but considering blanks to be junk. That prevents `' abcd'` from matching the `' abcd'` at the tail end of the second sequence directly. Instead only the `'abcd'` can match, and matches the leftmost `'abcd'` in the second sequence :

```
>>> s = SequenceMatcher(lambda x: x==" ", " abcd", "abcd abcd")
>>> s.find_longest_match(0, 5, 0, 9)
Match(a=1, b=0, size=4)
```

If no blocks match, this returns `(alo, blo, 0)`.

This method returns a *named tuple* `Match(a, b, size)`.

Modifié dans la version 3.9 : Added default arguments.

get_matching_blocks()

Return list of triples describing non-overlapping matching subsequences. Each triple is of the form `(i, j, n)`, and means that `a[i:i+n] == b[j:j+n]`. The triples are monotonically increasing in *i* and *j*.

The last triple is a dummy, and has the value `(len(a), len(b), 0)`. It is the only triple with `n == 0`. If `(i, j, n)` and `(i', j', n')` are adjacent triples in the list, and the second is not the last triple in the list, then `i+n < i'` or `j+n < j'`; in other words, adjacent triples always describe non-adjacent equal blocks.

```
>>> s = SequenceMatcher(None, "abxcd", "abcd")
>>> s.get_matching_blocks()
[Match(a=0, b=0, size=2), Match(a=3, b=2, size=2), Match(a=5, b=4, size=0)]
```

get_opcodes()

Return list of 5-tuples describing how to turn *a* into *b*. Each tuple is of the form `(tag, i1, i2, j1, j2)`. The first tuple has `i1 == j1 == 0`, and remaining tuples have *i1* equal to the *i2* from the preceding tuple, and, likewise, *j1* equal to the previous *j2*.

The *tag* values are strings, with these meanings :

Valeur	Signification
`'replace'`	`a[i1:i2]` should be replaced by `b[j1:j2]`.
`'delete'`	`a[i1:i2]` should be deleted. Note that `j1 == j2` in this case.
`'insert'`	`b[j1:j2]` should be inserted at `a[i1:i1]`. Note that `i1 == i2` in this case.
`'equal'`	`a[i1:i2] == b[j1:j2]` (the sub-sequences are equal).

Par exemple :

```
>>> a = "qabxcd"
>>> b = "abycdf"
>>> s = SequenceMatcher(None, a, b)
>>> for tag, i1, i2, j1, j2 in s.get_opcodes():
...     print('{:7}   a[{}:{}] --> b[{}:{}] {!r:>8} --> {!r}'.format(
...         tag, i1, i2, j1, j2, a[i1:i2], b[j1:j2]))
delete    a[0:1] --> b[0:0]      'q' --> ''
equal     a[1:3] --> b[0:2]     'ab' --> 'ab'
replace   a[3:4] --> b[2:3]      'x' --> 'y'
equal     a[4:6] --> b[3:5]     'cd' --> 'cd'
insert    a[6:6] --> b[5:6]       '' --> 'f'
```

get_grouped_opcodes(*n=3*)

Return a *generator* of groups with up to *n* lines of context.

Starting with the groups returned by `get_opcodes()`, this method splits out smaller change clusters and eliminates intervening ranges which have no changes.

The groups are returned in the same format as `get_opcodes()`.

ratio()

Return a measure of the sequences' similarity as a float in the range [0, 1].

Where T is the total number of elements in both sequences, and M is the number of matches, this is 2.0*M / T. Note that this is `1.0` if the sequences are identical, and `0.0` if they have nothing in common.

This is expensive to compute if `get_matching_blocks()` or `get_opcodes()` hasn't already been called, in which case you may want to try `quick_ratio()` or `real_quick_ratio()` first to get an upper bound.

Note

Caution : The result of a `ratio()` call may depend on the order of the arguments. For instance :

```
>>> SequenceMatcher(None, 'tide', 'diet').ratio()
0.25
>>> SequenceMatcher(None, 'diet', 'tide').ratio()
0.5
```

quick_ratio()

Return an upper bound on `ratio()` relatively quickly.

real_quick_ratio()

Return an upper bound on `ratio()` very quickly.

The three methods that return the ratio of matching to total characters can give different results due to differing levels of approximation, although `quick_ratio()` and `real_quick_ratio()` are always at least as large as `ratio()` :

```
>>> s = SequenceMatcher(None, "abcd", "bcde")
>>> s.ratio()
```

(suite sur la page suivante)

(suite de la page précédente)

```
0.75
>>> s.quick_ratio()
0.75
>>> s.real_quick_ratio()
1.0
```

6.3.2 SequenceMatcher Examples

This example compares two strings, considering blanks to be "junk" :

```
>>> s = SequenceMatcher(lambda x: x == " ",
...                     "private Thread currentThread;",
...                     "private volatile Thread currentThread;")
```

`ratio()` returns a float in [0, 1], measuring the similarity of the sequences. As a rule of thumb, a `ratio()` value over 0.6 means the sequences are close matches :

```
>>> print(round(s.ratio(), 3))
0.866
```

If you're only interested in where the sequences match, `get_matching_blocks()` is handy :

```
>>> for block in s.get_matching_blocks():
...     print("a[%d] and b[%d] match for %d elements" % block)
a[0] and b[0] match for 8 elements
a[8] and b[17] match for 21 elements
a[29] and b[38] match for 0 elements
```

Note that the last tuple returned by `get_matching_blocks()` is always a dummy, `(len(a), len(b), 0)`, and this is the only case in which the last tuple element (number of elements matched) is `0`.

If you want to know how to change the first sequence into the second, use `get_opcodes()` :

```
>>> for opcode in s.get_opcodes():
...     print("%6s a[%d:%d] b[%d:%d]" % opcode)
 equal a[0:8] b[0:8]
insert a[8:8] b[8:17]
 equal a[8:29] b[17:38]
```

Voir aussi

— The `get_close_matches()` function in this module which shows how simple code building on `SequenceMatcher` can be used to do useful work.
— Simple version control recipe for a small application built with `SequenceMatcher`.

6.3.3 Differ Objects

Note that `Differ`-generated deltas make no claim to be **minimal** diffs. To the contrary, minimal diffs are often counter-intuitive, because they synch up anywhere possible, sometimes accidental matches 100 pages apart. Restricting synch points to contiguous matches preserves some notion of locality, at the occasional cost of producing a longer diff.

The `Differ` class has this constructor :

class difflib.**Differ**(*linejunk=None*, *charjunk=None*)

Optional keyword parameters *linejunk* and *charjunk* are for filter functions (or `None`) :

linejunk : A function that accepts a single string argument, and returns true if the string is junk. The default is `None`, meaning that no line is considered junk.

charjunk : A function that accepts a single character argument (a string of length 1), and returns true if the character is junk. The default is `None`, meaning that no character is considered junk.

These junk-filtering functions speed up matching to find differences and do not cause any differing lines or characters to be ignored. Read the description of the `find_longest_match()` method's *isjunk* parameter for an explanation.

`Differ` objects are used (deltas generated) via a single method :

compare(*a*, *b*)

Compare two sequences of lines, and generate the delta (a sequence of lines).

Each sequence must contain individual single-line strings ending with newlines. Such sequences can be obtained from the `readlines()` method of file-like objects. The delta generated also consists of newline-terminated strings, ready to be printed as-is via the `writelines()` method of a file-like object.

6.3.4 Differ Example

This example compares two texts. First we set up the texts, sequences of individual single-line strings ending with newlines (such sequences can also be obtained from the `readlines()` method of file-like objects) :

```
>>> text1 = '''  1. Beautiful is better than ugly.
...   2. Explicit is better than implicit.
...   3. Simple is better than complex.
...   4. Complex is better than complicated.
... '''.splitlines(keepends=True)
>>> len(text1)
4
>>> text1[0][-1]
'\n'
>>> text2 = '''  1. Beautiful is better than ugly.
...   3.   Simple is better than complex.
...   4. Complicated is better than complex.
...   5. Flat is better than nested.
... '''.splitlines(keepends=True)
```

Next we instantiate a Differ object :

```
>>> d = Differ()
```

Note that when instantiating a `Differ` object we may pass functions to filter out line and character "junk." See the `Differ()` constructor for details.

Finally, we compare the two :

```
>>> result = list(d.compare(text1, text2))
```

`result` is a list of strings, so let's pretty-print it :

```
>>> from pprint import pprint
>>> pprint(result)
['    1. Beautiful is better than ugly.\n',
 '-   2. Explicit is better than implicit.\n',
 '-   3. Simple is better than complex.\n',
 '+   3.   Simple is better than complex.\n',
 '?     ++\n',
 '-   4. Complex is better than complicated.\n',
 '?            ^                     ---- ^\n',
```

(suite sur la page suivante)

(suite de la page précédente)

```
 '+   4. Complicated is better than complex.\n',
 '?           ++++ ^                      ^\n',
 '+   5. Flat is better than nested.\n']
```

As a single multi-line string it looks like this :

```
>>> import sys
>>> sys.stdout.writelines(result)
    1. Beautiful is better than ugly.
-   2. Explicit is better than implicit.
-   3. Simple is better than complex.
+   3.   Simple is better than complex.
?     ++
-   4. Complex is better than complicated.
?            ^                     ---- ^
+   4. Complicated is better than complex.
?           ++++ ^                      ^
+   5. Flat is better than nested.
```

6.3.5 A command-line interface to difflib

This example shows how to use difflib to create a `diff`-like utility.

```
""" Command line interface to difflib.py providing diffs in four formats:

* ndiff:    lists every line and highlights interline changes.
* context:  highlights clusters of changes in a before/after format.
* unified:  highlights clusters of changes in an inline format.
* html:     generates side by side comparison with change highlights.

"""

import sys, os, difflib, argparse
from datetime import datetime, timezone

def file_mtime(path):
    t = datetime.fromtimestamp(os.stat(path).st_mtime,
                               timezone.utc)
    return t.astimezone().isoformat()

def main():

    parser = argparse.ArgumentParser()
    parser.add_argument('-c', action='store_true', default=False,
                        help='Produce a context format diff (default)')
    parser.add_argument('-u', action='store_true', default=False,
                        help='Produce a unified format diff')
    parser.add_argument('-m', action='store_true', default=False,
                        help='Produce HTML side by side diff '
                             '(can use -c and -l in conjunction)')
    parser.add_argument('-n', action='store_true', default=False,
                        help='Produce a ndiff format diff')
    parser.add_argument('-l', '--lines', type=int, default=3,
                        help='Set number of context lines (default 3)')
    parser.add_argument('fromfile')
    parser.add_argument('tofile')
```

(suite sur la page suivante)

(suite de la page précédente)

```
    options = parser.parse_args()

    n = options.lines
    fromfile = options.fromfile
    tofile = options.tofile

    fromdate = file_mtime(fromfile)
    todate = file_mtime(tofile)
    with open(fromfile) as ff:
        fromlines = ff.readlines()
    with open(tofile) as tf:
        tolines = tf.readlines()

    if options.u:
        diff = difflib.unified_diff(fromlines, tolines, fromfile, tofile, fromdate,
↪ todate, n=n)
    elif options.n:
        diff = difflib.ndiff(fromlines, tolines)
    elif options.m:
        diff = difflib.HtmlDiff().make_file(fromlines,tolines,fromfile,tofile,
↪context=options.c,numlines=n)
    else:
        diff = difflib.context_diff(fromlines, tolines, fromfile, tofile, fromdate,
↪ todate, n=n)

    sys.stdout.writelines(diff)

if __name__ == '__main__':
    main()
```

6.3.6 ndiff example

This example shows how to use `difflib.ndiff()`.

```
"""ndiff [-q] file1 file2
    or
ndiff (-r1 | -r2) < ndiff_output > file1_or_file2

Print a human-friendly file difference report to stdout.  Both inter-
and intra-line differences are noted.  In the second form, recreate file1
(-r1) or file2 (-r2) on stdout, from an ndiff report on stdin.

In the first form, if -q ("quiet") is not specified, the first two lines
of output are

-: file1
+: file2

Each remaining line begins with a two-letter code:

    "- "    line unique to file1
    "+ "    line unique to file2
    "  "    line common to both files
    "? "    line not present in either input file

Lines beginning with "? " attempt to guide the eye to intraline
```

(suite sur la page suivante)

(suite de la page précédente)

```
differences, and were not present in either input file.  These lines can be
confusing if the source files contain tab characters.

The first file can be recovered by retaining only lines that begin with
"  " or "- ", and deleting those 2-character prefixes; use ndiff with -r1.

The second file can be recovered similarly, but by retaining only "  " and
"+ " lines; use ndiff with -r2; or, on Unix, the second file can be
recovered by piping the output through

    sed -n '/^[+ ] /s/^..//p'
"""

__version__ = 1, 7, 0

import difflib, sys

def fail(msg):
    out = sys.stderr.write
    out(msg + "\n\n")
    out(__doc__)
    return 0

# open a file & return the file object; gripe and return 0 if it
# couldn't be opened
def fopen(fname):
    try:
        return open(fname)
    except IOError as detail:
        return fail("couldn't open " + fname + ": " + str(detail))

# open two files & spray the diff to stdout; return false iff a problem
def fcompare(f1name, f2name):
    f1 = fopen(f1name)
    f2 = fopen(f2name)
    if not f1 or not f2:
        return 0

    a = f1.readlines(); f1.close()
    b = f2.readlines(); f2.close()
    for line in difflib.ndiff(a, b):
        print(line, end=' ')

    return 1

# crack args (sys.argv[1:] is normal) & compare;
# return false iff a problem

def main(args):
    import getopt
    try:
        opts, args = getopt.getopt(args, "qr:")
    except getopt.error as detail:
        return fail(str(detail))
    noisy = 1
    qseen = rseen = 0
```

(suite sur la page suivante)

(suite de la page précédente)

```
    for opt, val in opts:
        if opt == "-q":
            qseen = 1
            noisy = 0
        elif opt == "-r":
            rseen = 1
            whichfile = val
    if qseen and rseen:
        return fail("can't specify both -q and -r")
    if rseen:
        if args:
            return fail("no args allowed with -r option")
        if whichfile in ("1", "2"):
            restore(whichfile)
            return 1
        return fail("-r value must be 1 or 2")
    if len(args) != 2:
        return fail("need 2 filename args")
    f1name, f2name = args
    if noisy:
        print('-:', f1name)
        print('+:', f2name)
    return fcompare(f1name, f2name)

# read ndiff output from stdin, and print file1 (which=='1') or
# file2 (which=='2') to stdout

def restore(which):
    restored = difflib.restore(sys.stdin.readlines(), which)
    sys.stdout.writelines(restored)

if __name__ == '__main__':
    main(sys.argv[1:])
```

6.4 `textwrap` --- Text wrapping and filling

Source code : Lib/textwrap.py

Le module `textwrap` fournit quelques fonctions pratiques, comme `TextWrapper`, la classe qui fait tout le travail. Si vous ne faites que formater ou ajuster une ou deux chaînes de texte, les fonctions de commodité devraient être assez bonnes ; sinon, nous vous conseillons d'utiliser une instance de `TextWrapper` pour une meilleure efficacité.

`textwrap.`**`wrap`**(*text*, *width=70*, **, initial_indent=''*, *subsequent_indent=''*, *expand_tabs=True*, *replace_whitespace=True*, *fix_sentence_endings=False*, *break_long_words=True*, *drop_whitespace=True*, *break_on_hyphens=True*, *tabsize=8*, *max_lines=None*, *placeholder=' [...]'*)

Reformate le paragraphe simple dans *text* (une chaîne de caractères) de sorte que chaque ligne ait au maximum *largeur* caractères. Renvoie une liste de lignes de sortie, sans ligne vide ou caractère de fin de ligne à la fin.

Optional keyword arguments correspond to the instance attributes of `TextWrapper`, documented below.

Consultez la méthode `TextWrapper.wrap()` pour plus de détails sur le comportement de `wrap()`.

`textwrap.`**`fill`**(*text*, *width=70*, **, initial_indent=''*, *subsequent_indent=''*, *expand_tabs=True*, *replace_whitespace=True*, *fix_sentence_endings=False*, *break_long_words=True*, *drop_whitespace=True*, *break_on_hyphens=True*, *tabsize=8*, *max_lines=None*, *placeholder=' [...]'*)

Formate le paragraphe unique dans *text* et renvoie une seule chaîne dont le contenu est le paragraphe formaté. `fill()` est un raccourci pour

```
"\n".join(wrap(text, ...))
```

En particulier, `fill()` accepte exactement les mêmes arguments nommés que `wrap()`.

textwrap.**shorten**(*text*, *width*, **, *fix_sentence_endings=False*, *break_long_words=True*, *break_on_hyphens=True*, *placeholder=' [...]'*)

Réduit et tronque le *text* donné pour l'adapter à la *largeur* donnée.

First the whitespace in *text* is collapsed (all whitespace is replaced by single spaces). If the result fits in the *width*, it is returned. Otherwise, enough words are dropped from the end so that the remaining words plus the *placeholder* fit within *width* :

```
>>> textwrap.shorten("Hello  world!", width=12)
'Hello world!'
>>> textwrap.shorten("Hello  world!", width=11)
'Hello [...]'
>>> textwrap.shorten("Hello world", width=10, placeholder="...")
'Hello...'
```

Les arguments nommés optionnels correspondent aux attributs d'instance de `TextWrapper`, documentés ci-dessous. Notez que l'espace blanc est réduit avant que le texte ne soit passé à la fonction `fill()` de `TextWrapper`, donc changer la valeur de `tabsize`, `expand_tabs`, `drop_whitespace`, et `replace_whitespace` est sans effet.

Ajouté dans la version 3.4.

textwrap.**dedent**(*text*)

Supprime toutes les espaces en de tête de chaque ligne dans *text*.

Ceci peut être utilisé pour aligner les chaînes à trois guillemets avec le bord gauche de l'affichage, tout en les présentant sous forme indentée dans le code source.

Notez que les tabulations et les espaces sont traitées comme des espaces, mais qu'elles ne sont pas égales : les lignes `" hello"` et `"\thello"` sont considérées comme n'ayant pas d'espaces de tête communes.

Les lignes contenant uniquement des espaces sont ignorées en entrée et réduites à un seul caractère de saut de ligne en sortie.

Par exemple :

```
def test():
    # end first line with \ to avoid the empty line!
    s = '''\
    hello
      world
    '''
    print(repr(s))          # prints '    hello\n      world\n    '
    print(repr(dedent(s)))  # prints 'hello\n  world\n'
```

textwrap.**indent**(*text*, *prefix*, *predicate=None*)

Ajoute *prefix* au début des lignes sélectionnées dans *text*.

Les lignes sont séparées en appelant `text.splitlines(True)`.

Par défaut, *prefix* est ajouté à toutes les lignes qui ne sont pas constituées uniquement d'espaces (y compris les fins de ligne).

Par exemple :

```
>>> s = 'hello\n\n \nworld'
>>> indent(s, '  ')
'  hello\n\n \n  world'
```

L'argument optionnel *predicate* peut être utilisé pour contrôler quelles lignes sont en retrait. Par exemple, il est facile d'ajouter *prefix* aux lignes vides et aux lignes blanches seulement :

```
>>> print(indent(s, '+ ', lambda line: True))
+ hello
+
+
+ world
```

Ajouté dans la version 3.3.

`wrap()`, `fill()` et `shorten()` travaillent en créant une instance `TextWrapper` et en appelant une méthode unique sur celle-ci. Cette instance n'est pas réutilisée, donc pour les applications qui traitent plusieurs chaînes de texte en utilisant `wrap()` et/ou `fill()`, il peut être plus efficace de créer votre propre objet `TextWrapper`.

Le formatage du texte s'effectue en priorité sur les espaces puis juste après les traits d'union dans des mots séparés par des traits d'union ; ce n'est qu'alors que les mots longs seront cassés si nécessaire, à moins que `TextWrapper.break_long_words` soit défini sur *False*.

class textwrap.**TextWrapper**(***kwargs*)

Le constructeur `TextWrapper` accepte un certain nombre d'arguments nommés optionnels. Chaque argument nommé correspond à un attribut d'instance, donc par exemple

```
wrapper = TextWrapper(initial_indent="* ")
```

est identique à

```
wrapper = TextWrapper()
wrapper.initial_indent = "* "
```

You can reuse the same `TextWrapper` object many times, and you can change any of its options through direct assignment to instance attributes between uses.

Les attributs d'instance de la classe `TextWrapper` (et les arguments nommés au constructeur) sont les suivants :

width

(par défaut : `70`) Longueur maximale des lignes reformatées. Tant qu'il n'y a pas de mots individuels dans le texte d'entrée plus longs que `width`, `TextWrapper` garantit qu'aucune ligne de sortie n'est plus longue que `width` caractères.

expand_tabs

(default : `True`) If true, then all tab characters in *text* will be expanded to spaces using the `expandtabs()` method of *text*.

tabsize

(par défaut : `8`) Si `expand_tabs` est vrai, alors tous les caractères de tabulation dans *text* sont transformés en zéro ou plus d'espaces, selon la colonne courante et la taille de tabulation donnée.

Ajouté dans la version 3.3.

replace_whitespace

(par défaut : `True`) Si vrai, après l'expansion des tabulations mais avant le formatage, la méthode `wrap()` remplace chaque *blanc* par une espace unique. Les *blancs* remplacés sont les suivants : tabulation, nouvelle ligne, tabulation verticale, saut de page et retour chariot (`'\t\n\v\f\r'`).

> **Note**
>
> Si `expand_tabs` est *False* et `replace_whitespace` est vrai, chaque caractère de tabulation est remplacé par une espace unique, ce qui *n'est pas* la même chose que l'extension des tabulations.

> **Note**
>
> Si `replace_whitespace` est faux, de nouvelles lignes peuvent apparaître au milieu d'une ligne et provoquer une sortie étrange. Pour cette raison, le texte doit être divisé en paragraphes (en utilisant `str.splitlines()` ou similaire) qui sont formatés séparément.

drop_whitespace

(par défaut : `True`) Si *True*, l'espace au début et à la fin de chaque ligne (après le formatage mais avant l'indentation) est supprimée. L'espace au début du paragraphe n'est cependant pas supprimée si elle n'est pas suivie par une espace. Si les espaces en cours de suppression occupent une ligne entière, la ligne entière est supprimée.

initial_indent

(par défaut : `''`) Chaîne qui est ajoutée à la première ligne de la sortie formatée. Elle compte pour le calcul de la longueur de la première ligne. La chaîne vide n'est pas indentée.

subsequent_indent

(par défaut : `''`) Chaîne qui préfixe toutes les lignes de la sortie formatée sauf la première. Elle compte pour le calcul de la longueur de chaque ligne à l'exception de la première.

fix_sentence_endings

(default : `False`) If true, `TextWrapper` attempts to detect sentence endings and ensure that sentences are always separated by exactly two spaces. This is generally desired for text in a monospaced font. However, the sentence detection algorithm is imperfect : it assumes that a sentence ending consists of a lowercase letter followed by one of `'.'`, `'!'`, or `'?'`, possibly followed by one of `'"'` or `"'"`, followed by a space. One problem with this algorithm is that it is unable to detect the difference between "Dr." in

```
[...] Dr. Frankenstein's monster [...]
```

et "Spot." dans

```
[...] See Spot. See Spot run [...]
```

`fix_sentence_endings` est *False* par défaut.

Étant donné que l'algorithme de détection de phrases repose sur `string.lowercase` pour la définition de « lettres minuscules » et sur une convention consistant à utiliser deux espaces après un point pour séparer les phrases sur la même ligne, ceci est spécifique aux textes de langue anglaise.

break_long_words

(par défaut : `True`) Si *True*, alors les mots plus longs que `width` sont cassés afin de s'assurer qu'aucune ligne ne soit plus longue que `width`. Si c'est *False*, les mots longs ne sont pas cassés et certaines lignes peuvent être plus longues que `width` (les mots longs seront mis sur une ligne qui leur est propre, afin de minimiser le dépassement de `width`).

break_on_hyphens

(par défaut : `True`) Si c'est vrai, le formatage se fait de préférence sur les espaces et juste après sur les traits d'union des mots composés, comme il est d'usage en anglais. Si *False*, seuls les espaces sont considérées comme de bons endroits pour les sauts de ligne, mais vous devez définir `break_long_words` à *False* si vous voulez des mots vraiment insécables. Le comportement par défaut dans les versions précédentes était de toujours permettre de couper les mots avec trait d'union.

max_lines

(par défaut : `None`) Si ce n'est pas `None`, alors la sortie contient au maximum *max_lines* lignes, avec *placeholder* à la fin de la sortie.

Ajouté dans la version 3.4.

placeholder

(par défaut : `'' [...]'`) Chaîne qui apparaît à la fin du texte de sortie s'il a été tronqué.

Ajouté dans la version 3.4.

`TextWrapper` fournit également quelques méthodes publiques, analogues aux fonctions de commodité au niveau du module :

wrap(*text*)

Formate le paragraphe unique dans *text* (une chaîne de caractères) de sorte que chaque ligne ait au maximum `width` caractères. Toutes les options de formatage sont tirées des attributs d'instance de l'instance de classe `TextWrapper`. Renvoie une liste de lignes de sortie, sans nouvelles lignes finales. Si la sortie formatée n'a pas de contenu, la liste vide est renvoyée.

fill(*text*)

Formate le paragraphe unique de *text* et renvoie une seule chaîne contenant le paragraphe formaté.

6.5 `unicodedata` --- Unicode Database

This module provides access to the Unicode Character Database (UCD) which defines character properties for all Unicode characters. The data contained in this database is compiled from the UCD version 16.0.0.

The module uses the same names and symbols as defined by Unicode Standard Annex #44, "Unicode Character Database". It defines the following functions :

`unicodedata.`**lookup**(*name*)

Retrouver un caractère par nom. Si un caractère avec le nom donné est trouvé, renvoyer le caractère correspondant. S'il n'est pas trouvé, `KeyError` est levée.

Modifié dans la version 3.3 : La gestion des alias [1] et des séquences nommées [2] a été ajouté.

`unicodedata.`**name**(*chr*[, *default*])

Renvoie le nom assigné au caractère *chr* comme une chaîne de caractères. Si aucun nom n'est défini, *default* est renvoyé, ou, si ce dernier n'est pas renseigné `ValueError` est levée.

`unicodedata.`**decimal**(*chr*[, *default*])

Renvoie la valeur décimale assignée au caractère *chr* comme un entier. Si aucune valeur de ce type n'est définie, *default* est renvoyé, ou, si ce dernier n'est pas renseigné, `ValueError` est levée.

`unicodedata.`**digit**(*chr*[, *default*])

Renvoie le chiffre assigné au caractère *chr* comme un entier. Si aucune valeur de ce type n'est définie, *default* est renvoyé, ou, si ce dernier n'est pas renseigné, `ValueError` est levée.

`unicodedata.`**numeric**(*chr*[, *default*])

Renvoie la valeur numérique assignée au caractère *chr* comme un entier. Si aucune valeur de ce type n'est définie, *default* est renvoyé, ou, si ce dernier n'est pas renseigné, `ValueError` est levée.

`unicodedata.`**category**(*chr*)

Renvoie la catégorie générale assignée au caractère *chr* comme une chaîne de caractères.

`unicodedata.`**bidirectional**(*chr*)

Renvoie la classe bidirectionnelle assignée au caractère *chr* comme une chaîne de caractères. Si aucune valeur de ce type n'est définie, une chaîne de caractères vide est renvoyée.

1. https://www.unicode.org/Public/16.0.0/ucd/NameAliases.txt
2. https://www.unicode.org/Public/16.0.0/ucd/NamedSequences.txt

`unicodedata.combining(chr)`

Renvoie la classe de combinaison canonique assignée au caractère *chr* comme un entier. Envoie `0` si aucune classe de combinaison n'est définie.

`unicodedata.east_asian_width(chr)`

Renvoie la largeur est-asiatique assignée à un caractère *chr* comme une chaîne de caractères.

`unicodedata.mirrored(chr)`

Renvoie la propriété miroir assignée au caractère *chr* comme un entier. Renvoie `1` si le caractère a été identifié comme un caractère "réfléchi" dans du texte bidirectionnel, sinon `0`.

`unicodedata.decomposition(chr)`

Renvoie le tableau associatif de décomposition de caractère assigné au caractère *chr* comme une chaîne de caractères. Une chaîne de caractère vide est renvoyée dans le cas où aucun tableau associatif de ce type n'est défini.

`unicodedata.normalize(form, unistr)`

Renvoie la forme normale *form* de la chaîne de caractère Unicode *unistr*. Les valeurs valides de *form* sont *NFC*, *NFKC*, *NFD*, et *NFKD*.

Le standard Unicode définit les différentes variantes de normalisation d'une chaîne de caractères Unicode en se basant sur les définitions d'équivalence canonique et d'équivalence de compatibilité. En Unicode, plusieurs caractères peuvent être exprimés de différentes façons. Par exemple, le caractère *U+00C7* (*LATIN CAPITAL LETTER C WITH CEDILLA*) peut aussi être exprimé comme la séquence *U+0043* (*LATIN CAPITAL LETTER C*) *U+0327* (*COMBINING CEDILLA*).

Pour chaque caractère, il existe deux formes normales : la forme normale C et la forme normale D. La forme normale D (NFD) est aussi appelée décomposition canonique, et traduit chaque caractère dans sa forme décomposée. La forme normale C (NFC) applique d'abord la décomposition canonique, puis compose à nouveau les caractères pré-combinés.

En plus de ces deux formes, il existe deux formes nominales basées sur l'équivalence de compatibilité. En Unicode, certains caractères sont gérés alors qu'ils sont normalement unifiés avec d'autres caractères. Par exemple, U+2160 (ROMAN NUMERAL ONE) est vraiment la même chose que U+0049 (LATIN CAPITAL LETTER I). Cependant, ce caractère est supporté par souci de compatibilité avec les jeux de caractères existants (par exemple *gb2312*).

La forme normale KD (NFKD) applique la décomposition de compatibilité, c'est-à-dire remplacer les caractères de compatibilités avec leurs équivalents. La forme normale KC (NFKC) applique d'abord la décomposition de compatibilité, puis applique la composition canonique.

Même si deux chaînes de caractères Unicode sont normalisées et ont la même apparence pour un lecteur humain , si un a des caractères combinés et l'autre n'en a pas, elles peuvent ne pas être égales lors d'une comparaison.

`unicodedata.is_normalized(form, unistr)`

Return whether the Unicode string *unistr* is in the normal form *form*. Valid values for *form* are 'NFC', 'NFKC', 'NFD', and 'NFKD'.

Ajouté dans la version 3.8.

De plus, ce module expose la constante suivante :

`unicodedata.unidata_version`

La version de la base de données Unicode utilisée dans ce module.

`unicodedata.ucd_3_2_0`

Ceci est un objet qui a les mêmes méthodes que le module, mais qui utilise la version 3.2 de la base de données Unicode, pour les applications qui nécessitent cette version spécifique de la base de données Unicode (comme l'IDNA).

Exemples :

```
>>> import unicodedata
>>> unicodedata.lookup('LEFT CURLY BRACKET')
'{'
>>> unicodedata.name('/')
'SOLIDUS'
>>> unicodedata.decimal('9')
9
>>> unicodedata.decimal('a')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: not a decimal
>>> unicodedata.category('A')  # 'L'etter, 'u'ppercase
'Lu'
>>> unicodedata.bidirectional('\u0660') # 'A'rabic, 'N'umber
'AN'
```

Notes

6.6 `stringprep` --- Internet String Preparation

Code source : Lib/stringprep.py

Nommer les différentes choses d'internet (comme les hôtes) amène souvent au besoin de comparer ces identifiants, ce qui nécessite un critère d'« égalité ». La manière dont cette comparaison est effectuée dépend du domaine d'application, c'est-à-dire si elle doit être sensible à la casse ou non. Il peut être aussi nécessaire de restreindre les identifiants possibles, pour permettre uniquement les identifiants composés de caractères « imprimables ».

La **RFC 3454** définit une procédure pour "préparer" des chaines de caractères Unicode dans les protocoles internet. Avant de passer des chaines de caractères sur le câble, elles sont traitées avec la procédure de préparation, après laquelle ils obtiennent une certaines forme normalisée. Les RFC définissent un lot de tables, qui peuvent être combinées en profils. Chaque profil doit définir quelles tables il utilise et quelles autres parties optionnelles de la procédure *stringprep* font partie du profil. Un exemple de profil *stringprep* est *nameprep*, qui est utilisé pour les noms de domaine internationalisés.

The module `stringprep` only exposes the tables from **RFC 3454**. As these tables would be very large to represent as dictionaries or lists, the module uses the Unicode character database internally. The module source code itself was generated using the `mkstringprep.py` utility.

En conséquence, ces tables sont exposées en tant que fonctions et non en structures de données. Il y a deux types de tables dans la RFC : les ensemble et les mises en correspondance. Pour un ensemble, `stringprep` fournit la "fonction caractéristique", c'est-à-dire une fonction qui renvoie `True` si le paramètre fait partie de l'ensemble. Pour les mises en correspondance, il fournit la fonction de mise en correspondance : ayant obtenu la clé, il renvoie la valeur associée. Ci-dessous se trouve une liste de toutes les fonctions disponibles dans le module.

`stringprep.`**`in_table_a1`**(*code*)

Détermine si le code est en table A.1 (points de code non-assigné dans Unicode 3.2).

`stringprep.`**`in_table_b1`**(*code*)

Détermine si le code est en table B.1 (habituellement mis en correspondance avec rien).

`stringprep.`**`map_table_b2`**(*code*)

Renvoie la valeur correspondante à *code* selon la table B.2 (mise en correspondance pour la gestion de la casse utilisée avec *NFKC*).

`stringprep.`**`map_table_b3`**(*code*)

Renvoie la valeur correspondante à *code* dans la table B.3 (mise en correspondance pour la gestion de la casse utilisée sans normalisation).

`stringprep.`**`in_table_c11`**(*code*)

Détermine si le code est dans la table C.1.1 (caractères d'espacement ASCII).

`stringprep.`**`in_table_c12`**(*code*)

Détermine si le code est dans la table C.1.2 (caractères d'espacement non ASCII).

`stringprep.`**`in_table_c11_c12`**(*code*)

Détermine si le code est dans la table C.1 (caractères d'espacement, union de C.1.1 et C.1.2).

`stringprep.`**`in_table_c21`**(*code*)

Détermine si le code est dans la table C.2.1 (caractères de contrôle ASCII).

`stringprep.`**`in_table_c22`**(*code*)

Détermine si le code est en table C.2.2 (caractères de contrôle non ASCII).

`stringprep.`**`in_table_c21_c22`**(*code*)

Détermine si le code est dans la table C.2 (caractères de contrôle, union de C.2.1 et C.2.2).

`stringprep.`**`in_table_c3`**(*code*)

Détermine si le code est en table C.3 (usage privé).

`stringprep.`**`in_table_c4`**(*code*)

Détermine si le code est dans la table C.4 (points de code non-caractère).

`stringprep.`**`in_table_c5`**(*code*)

Détermine si le code est en table C.5 (codes substituts).

`stringprep.`**`in_table_c6`**(*code*)

Détermine si le code est dans la table C.6 (Inapproprié pour texte brut).

`stringprep.`**`in_table_c7`**(*code*)

Détermine si le code est dans la table C.7 (inapproprié pour les représentations *canonics1*).

`stringprep.`**`in_table_c8`**(*code*)

Détermine si le code est dans la table C.8 (change de propriétés d'affichage ou sont obsolètes).

`stringprep.`**`in_table_c9`**(*code*)

Détermine si le code est dans la table C.9 (caractères de marquage).

`stringprep.`**`in_table_d1`**(*code*)

Détermine si le code est en table D.1 (caractères avec propriété bidirectionnelle "R" ou "AL").

`stringprep.`**`in_table_d2`**(*code*)

Détermine si le code est dans la table D.2 (caractères avec propriété bidirectionnelle "L").

6.7 `readline` --- GNU readline interface

Le module `readline` définit des fonctions pour faciliter la complétion et la lecture/écriture des fichiers d'historique depuis l'interpréteur Python. Ce module peut être utilisé directement, ou depuis le module `rlcompleter`, qui gère la complétion des mots clefs dans l'invite de commande interactive. Les paramétrages faits en utilisant ce module affectent à la fois le comportement de l'invite de commande interactive et l'invite de commande fournie par la fonction native `input()`.

Readline keybindings may be configured via an initialization file, typically `.inputrc` in your home directory. See Readline Init File in the GNU Readline manual for information about the format and allowable constructs of that file, and the capabilities of the Readline library in general.

Availability : not Android, not iOS, not WASI.

This module is not supported on *mobile platforms* or *WebAssembly platforms*.

Note

The underlying Readline library API may be implemented by the `editline` (`libedit`) library instead of GNU readline. On macOS the *`readline`* module detects which library is being used at run time.

The configuration file for `editline` is different from that of GNU readline. If you programmatically load configuration strings you can use *`backend`* to determine which library is being used.

If you use `editline`/`libedit` readline emulation on macOS, the initialization file located in your home directory is named `.editrc`. For example, the following content in `~/.editrc` will turn ON *vi* keybindings and TAB completion :

```
python:bind -v
python:bind ^I rl_complete
```

Also note that different libraries may use different history file formats. When switching the underlying library, existing history files may become unusable.

`readline.`**`backend`**

The name of the underlying Readline library being used, either `"readline"` or `"editline"`.

Ajouté dans la version 3.13.

6.7.1 Fichier d'initialisation

Les fonctions suivantes se rapportent au fichier d'initialisation et à la configuration utilisateur :

`readline.`**`parse_and_bind`**(*string*)

Exécute la ligne d'initialisation fournie dans l'argument *string*. Cela appelle la fonction `rl_parse_and_bind()` de la bibliothèque sous-jacente.

`readline.`**`read_init_file`**([*filename*])

Exécute un fichier d'initialisation *readline*. Le nom de fichier par défaut est le dernier nom de fichier utilisé. Cela appelle la fonction `rl_read_init_file()` de la bibliothèque sous-jacente.

6.7.2 Tampon de ligne

Les fonctions suivantes opèrent sur le tampon de ligne :

`readline.`**`get_line_buffer`**()

Renvoie le contenu courant du tampon de ligne (`rl_line_buffer` dans la bibliothèque sous-jacente).

`readline.`**`insert_text`**(*string*)

Insère du texte dans le tampon de ligne à la position du curseur. Cela appelle la fonction `rl_insert_text()` de la bibliothèque sous-jacente, mais ignore la valeur de retour.

`readline.`**`redisplay`**()

Change ce qui est affiché sur l'écran pour représenter le contenu courant de la ligne de tampon. Cela appelle la fonction `rl_redisplay()` dans la bibliothèque sous-jacente.

6.7.3 Fichier d'historique

les fonctions suivantes opèrent sur un fichier d'historique :

`readline.`**`read_history_file`**([*filename*])

Charge un fichier d'historique de *readline*, et l'ajoute à la liste d'historique. Le fichier par défaut est `~/.history`. Cela appelle la fonction `read_history()` de la bibliothèque sous-jacente.

`readline.`**`write_history_file`**([*filename*])

Enregistre la liste de l'historique dans un fichier d'historique de *readline*, en écrasant un éventuel fichier existant. Le nom de fichier par défaut est `~/.history`. Cela appelle la fonction `write_history()` de la bibliothèque sous-jacente.

readline.**append_history_file**(*nelements*[, *filename*])

Ajoute les derniers objets *nelements* de l'historique dans un fichier. Le nom de fichier par défaut est `~/.history`. Le fichier doit déjà exister. Cela appelle la fonction `append_history()` de la bibliothèque sous-jacente.

Ajouté dans la version 3.5.

readline.**get_history_length**()

readline.**set_history_length**(*length*)

Définit ou renvoie le nombre souhaité de lignes à enregistrer dans le fichier d'historique. La fonction *write_history_file()* utilise cette valeur pour tronquer le fichier d'historique, en appelant `history_truncate_file()` de la bibliothèque sous-jacente. Les valeurs négatives impliquent une taille de fichier d'historique illimitée.

6.7.4 Liste d'historique

Les fonctions suivantes opèrent sur une liste d'historique globale :

readline.**clear_history**()

Effacer l'historique courant. Cela appelle la fonction `clear_history()` de la bibliothèque sous-jacente. La fonction Python existe seulement si Python à été compilé pour une version de la bibliothèque qui le gère.

readline.**get_current_history_length**()

Renvoie le nombre d'objets actuellement dans l'historique. (C'est différent de *get_history_length()*, qui renvoie le nombre maximum de lignes qui vont être écrites dans un fichier d'historique.)

readline.**get_history_item**(*index*)

Renvoie le contenu courant de l'objet d'historique à *index*. L'index de l'objet commence à 1. Cela appelle `history_get()` de la bibliothèque sous-jacente.

readline.**remove_history_item**(*pos*)

Supprime l'objet de l'historique défini par sa position depuis l'historique. L'index de la position commence à zéro. Cela appelle la fonction `remove_history()` de la bibliothèque sous-jacente.

readline.**replace_history_item**(*pos*, *line*)

Remplace un objet de l'historique à la position définie par *line*. L'index de la position commence à zéro. Cela appelle `replace_history_entry()` de la bibliothèque sous-jacente.

readline.**add_history**(*line*)

Ajoute *line* au tampon d'historique, comme si c'était la dernière ligne saisie. Cela appelle la fonction `add_history()` de la libraire sous-jacente.

readline.**set_auto_history**(*enabled*)

Active ou désactive les appels automatiques à la fonction `add_history()` lors de la lecture d'une entrée via *readline*. L'argument *enabled* doit être une valeur booléenne qui lorsqu'elle est vraie, active l'historique automatique, et qui lorsqu'elle est fausse, désactive l'historique automatique.

Ajouté dans la version 3.6.

Particularité de l'implémentation CPython : Auto history is enabled by default, and changes to this do not persist across multiple sessions.

6.7.5 Fonctions de rappel au démarrage

readline.**set_startup_hook**([*function*])

Définit ou supprime la fonction invoquée par le la fonction de retour `rl_startup_hook` de la bibliothèque sous-jacente. Si *function* est spécifié, il est utilisé en tant que nouvelle fonction de rappel ; si omis ou `None`, toute fonction déjà installée est supprimée. La fonction de rappel est appelée sans arguments juste avant que *readline* affiche la première invite de commande.

`readline.set_pre_input_hook([function])`

Définit ou supprime la fonction invoquée par la fonction de retour `rl_pre_input_hook` de la bibliothèque sous-jacente. Si *function* est spécifié, il sera utilisé par la nouvelle fonction de rappel ; si omis ou `None`, toute fonction déjà installée est supprimée. La fonction de rappel est appelée sans arguments après que la première invite de commande ait été affichée et juste avant que *readline* commence à lire les caractères saisis. Cette fonction existe seulement si Python a été compilé pour une version de la bibliothèque qui le gère.

6.7.6 Complétion

Les fonctions suivantes relatent comment implémenter une fonction de complétion d'un mot spécifique. C'est typiquement déclenché par la touche Tab, et peut suggérer et automatiquement compléter un mot en cours de saisie. Par défaut, Readline est configuré pour être utilisé par `rlcompleter` pour compléter les mots clefs de Python pour l'interpréteur interactif. Si le module `readline` doit être utilisé avec une complétion spécifique, un ensemble de mots délimiteurs doivent être définis.

`readline.set_completer([function])`

Définit ou supprime la fonction de complétion. Si *function* est spécifié, il sera utilisé en tant que nouvelle fonction de complétion ; si omis ou `None`, toute fonction de complétion déjà installé est supprimé. La fonction de complétion est appelée telle que `function(text, state)`, pour *state* valant `0`, `1`, `2`, ..., jusqu'à ce qu'elle renvoie une valeur qui n'est pas une chaîne de caractères. Elle doit renvoyer les prochaines complétions possibles commençant par *text*.

La fonction de complétion installée est invoquée par la fonction de retour *entry_func* passée à `rl_completion_matches()` de la bibliothèque sous-jacente. La chaîne de caractère *text* va du premier paramètres vers la fonction de retour `rl_attempted_completion_function` de la bibliothèque sous-jacente.

`readline.get_completer()`

Récupère la fonction de complétion, ou `None` si aucune fonction de complétion n'a été définie.

`readline.get_completion_type()`

Récupère le type de complétion essayé. Cela renvoie la variable `rl_completion_type` dans la bibliothèque sous-jacente en tant qu'entier.

`readline.get_begidx()`

`readline.get_endidx()`

Get the beginning or ending index of the completion scope. These indexes are the *start* and *end* arguments passed to the `rl_attempted_completion_function` callback of the underlying library. The values may be different in the same input editing scenario based on the underlying C readline implementation. Ex : libedit is known to behave differently than libreadline.

`readline.set_completer_delims(string)`

`readline.get_completer_delims()`

Définit ou récupère les mots délimitants pour la complétion. Ceux-ci déterminent le début du mot devant être considéré pour la complétion (le contexte de la complétion). Ces fonctions accèdent à la variable `rl_completer_word_break_characters` de la bibliothèque sous-jacente.

`readline.set_completion_display_matches_hook([function])`

Définit ou supprime la fonction d'affichage de la complétion. Si *function* est spécifié, il sera utilisé en tant que nouvelle fonction d'affichage de complétion ; si omis ou `None`, toute fonction de complétion déjà installée est supprimée. Cela définit ou supprime la fonction de retour `rl_completion_display_matches_hook` dans la bibliothèque sous-jacente. La fonction d'affichage de complétion est appelée telle que `function(substitution, [matches], longest_match_length)` une seule fois lorsque les correspondances doivent être affichées.

6.7.7 Exemple

L'exemple suivant démontre comment utiliser les fonctions de lecture et d'écriture de l'historique du module `readline` pour charger ou sauvegarder automatiquement un fichier d'historique nommé `.python_history` depuis le répertoire d'accueil de l'utilisateur. Le code ci-dessous doit normalement être exécuté automatiquement durant une session interactive depuis le fichier de l'utilisateur `PYTHONSTARTUP`.

```
import atexit
import os
import readline

histfile = os.path.join(os.path.expanduser("~"), ".python_history")
try:
    readline.read_history_file(histfile)
    # default history len is -1 (infinite), which may grow unruly
    readline.set_history_length(1000)
except FileNotFoundError:
    pass

atexit.register(readline.write_history_file, histfile)
```

Ce code est en réalité automatiquement exécuté lorsque Python tourne en mode interactif (voir *Readline configuration*).

L'exemple suivant atteint le même objectif mais gère des sessions interactives concurrentes, en ajoutant seulement le nouvel historique.

```
import atexit
import os
import readline
histfile = os.path.join(os.path.expanduser("~"), ".python_history")

try:
    readline.read_history_file(histfile)
    h_len = readline.get_current_history_length()
except FileNotFoundError:
    open(histfile, 'wb').close()
    h_len = 0

def save(prev_h_len, histfile):
    new_h_len = readline.get_current_history_length()
    readline.set_history_length(1000)
    readline.append_history_file(new_h_len - prev_h_len, histfile)
atexit.register(save, h_len, histfile)
```

L'exemple suivant étend la classe `code.InteractiveConsole` pour gérer la sauvegarde/restauration de l'historique.

```
import atexit
import code
import os
import readline

class HistoryConsole(code.InteractiveConsole):
    def __init__(self, locals=None, filename="<console>",
                 histfile=os.path.expanduser("~/.console-history")):
        code.InteractiveConsole.__init__(self, locals, filename)
        self.init_history(histfile)

    def init_history(self, histfile):
        readline.parse_and_bind("tab: complete")
        if hasattr(readline, "read_history_file"):
            try:
                readline.read_history_file(histfile)
            except FileNotFoundError:
```

(suite sur la page suivante)

(suite de la page précédente)

```
                pass
            atexit.register(self.save_history, histfile)

    def save_history(self, histfile):
        readline.set_history_length(1000)
        readline.write_history_file(histfile)
```

6.8 rlcompleter --- Completion function for GNU readline

Code source : Lib/rlcompleter.py

The rlcompleter module defines a completion function suitable to be passed to *set_completer()* in the *readline* module.

When this module is imported on a Unix platform with the *readline* module available, an instance of the *Completer* class is automatically created and its *complete()* method is set as the *readline completer*. The method provides completion of valid Python identifiers and keywords.

Exemple :

```
>>> import rlcompleter
>>> import readline
>>> readline.parse_and_bind("tab: complete")
>>> readline. <TAB PRESSED>
readline.__doc__          readline.get_line_buffer(  readline.read_init_file(
readline.__file__         readline.insert_text(      readline.set_completer(
readline.__name__         readline.parse_and_bind(
>>> readline.
```

The rlcompleter module is designed for use with Python's interactive mode. Unless Python is run with the -S option, the module is automatically imported and configured (see *Readline configuration*).

Sur les plate-formes sans *readline*, la classe *Completer* définie par ce module peut quand même être utilisée pour des fins personnalisées.

class rlcompleter.**Completer**

Les objets pour la complétion (*Completer objects* en anglais) disposent de la méthode suivante :

complete(*text*, *state*)

Return the next possible completion for *text*.

When called by the *readline* module, this method is called successively with state == 0, 1, 2, ... until the method returns None.

Si *text* ne contient pas un caractère point ('.'), il puise dans les noms actuellement définis dans *__main__*, *builtins* ainsi que les mots-clés (ainsi que définis par le module *keyword*)

If called for a dotted name, it will try to evaluate anything without obvious side-effects (functions will not be evaluated, but it can generate calls to __getattr__()) up to the last part, and find matches for the rest via the *dir()* function. Any exception raised during the evaluation of the expression is caught, silenced and *None* is returned.

CHAPITRE 7

Services autour des Données Binaires

Les modules décrits dans ce chapitre fournissent des services élémentaires pour manipuler des données binaires. Les autres manipulations sur les données binaires, particulièrement celles en relation avec les formats de fichier et les protocoles réseaux sont décrits dans leurs propres chapitres.

Certaines bibliothèques décrites dans *Services de Manipulation de Texte* fonctionnent aussi avec soit des formats binaires compatibles ASCII (comme le module `re`) soit toutes les données binaires (comme le module `difflib`).

En complément, consultez la documentation des types natifs binaires dans *Séquences Binaires — bytes, bytearray, vue mémoire*.

7.1 `struct` --- Interpret bytes as packed binary data

Code source : Lib/struct.py

This module converts between Python values and C structs represented as Python `bytes` objects. Compact *format strings* describe the intended conversions to/from Python values. The module's functions and objects can be used for two largely distinct applications, data exchange with external sources (files or network connections), or data transfer between the Python application and the C layer.

> **Note**
>
> When no prefix character is given, native mode is the default. It packs or unpacks data based on the platform and compiler on which the Python interpreter was built. The result of packing a given C struct includes pad bytes which maintain proper alignment for the C types involved ; similarly, alignment is taken into account when unpacking. In contrast, when communicating data between external sources, the programmer is responsible for defining byte ordering and padding between elements. See *Boutisme, taille et alignement* for details.

Plusieurs fonctions de `struct` (et méthodes de `Struct`) prennent un argument *buffer*. Cet argument fait référence à des objets qui implémentent le protocole tampon et qui proposent un tampon soit en lecture seule, soit en lecture-écriture. Les types les plus courants qui utilisent cette fonctionnalité sont `bytes` et `bytearray`, mais beaucoup d'autres types qui peuvent être considérés comme des tableaux d'octets implémentent le protocole tampon ; ils peuvent ainsi être lus ou remplis depuis un objet `bytes` sans faire de copie.

7.1.1 Fonctions et exceptions

Le module définit les exceptions et fonctions suivantes :

exception `struct.error`

Exception levée à plusieurs occasions ; l'argument est une chaîne qui décrit ce qui ne va pas.

`struct.pack`(*format*, *v1*, *v2*, ...)

Renvoie un objet *bytes* contenant les valeurs *v1*, *v2*… agrégées conformément à la chaîne de format *format*. Les arguments doivent correspondre exactement aux valeurs requises par le format.

`struct.pack_into`(*format*, *buffer*, *offset*, *v1*, *v2*, ...)

Agrège les valeurs *v1*, *v2*… conformément à la chaîne de format *format* et écrit les octets agrégés dans le tampon *buffer*, en commençant à la position *offset*. Notez que *offset* est un argument obligatoire.

`struct.unpack`(*format*, *buffer*)

Dissocie depuis le tampon *buffer* (en supposant que celui-ci a été agrégé avec `pack(format, …)`) à l'aide de la chaîne de format *format*. Le résultat est un *n*-uplet, qui peut éventuellement ne contenir qu'un seul élément. La taille de *buffer* en octets doit correspondre à la taille requise par le format, telle que calculée par `calcsize()`.

`struct.unpack_from`(*format*, /, *buffer*, *offset=0*)

Dissocie les éléments du tampon *buffer*, en commençant à la position *offset*, conformément à la chaîne de format *format*. Le résultat est un n-uplet, qui peut éventuellement ne contenir qu'un seul élément. La taille du tampon en octets, en commençant à la position *offset*, doit être au moins égale à la taille requise par le format, telle que calculée par `calcsize()`.

`struct.iter_unpack`(*format*, *buffer*)

Iteratively unpack from the buffer *buffer* according to the format string *format*. This function returns an iterator which will read equally sized chunks from the buffer until all its contents have been consumed. The buffer's size in bytes must be a multiple of the size required by the format, as reflected by `calcsize()`.

Chaque itération produit un *n*-uplet tel que spécifié par la chaîne de format.

Ajouté dans la version 3.4.

`struct.calcsize`(*format*)

Renvoie la taille de la structure (et donc celle de l'objet *bytes* produit par `pack(format, ...)`) correspondant à la chaîne de format *format*.

7.1.2 Chaînes de spécification du format

Format strings describe the data layout when packing and unpacking data. They are built up from *format characters*, which specify the type of data being packed/unpacked. In addition, special characters control the *byte order, size and alignment*. Each format string consists of an optional prefix character which describes the overall properties of the data and one or more format characters which describe the actual data values and padding.

Boutisme, taille et alignement

By default, C types are represented in the machine's native format and byte order, and properly aligned by skipping pad bytes if necessary (according to the rules used by the C compiler). This behavior is chosen so that the bytes of a packed struct correspond exactly to the memory layout of the corresponding C struct. Whether to use native byte ordering and padding or standard formats depends on the application.

Cependant, le premier caractère de la chaîne de format peut être utilisé pour indiquer le boutisme, la taille et l'alignement des données agrégées, conformément à la table suivante :

Caractère	Boutisme	Taille	Alignement
@	natif	natif	natif
=	natif	standard	aucun
<	petit-boutiste	standard	aucun
>	gros-boutiste	standard	aucun
!	réseau (= gros-boutiste)	standard	aucun

Si le premier caractère n'est pas dans cette liste, le module se comporte comme si `'@'` avait été indiqué.

Note

The number 1023 (`0x3ff` in hexadecimal) has the following byte representations :

— `03 ff` in big-endian (>)

— `ff 03` in little-endian (<)

Python example :

```
>>> import struct
>>> struct.pack('>h', 1023)
b'\x03\xff'
>>> struct.pack('<h', 1023)
b'\xff\x03'
```

Native byte order is big-endian or little-endian, depending on the host system. For example, Intel x86, AMD64 (x86-64), and Apple M1 are little-endian ; IBM z and many legacy architectures are big-endian. Use `sys.byteorder` to check the endianness of your system.

La taille et l'alignement natifs sont déterminés en utilisant l'expression `sizeof` du compilateur C. Leur valeur est toujours combinée au boutisme natif.

La taille standard dépend seulement du caractère du format ; référez-vous au tableau dans la section *Caractères de format*.

Notez la différence entre `'@'` et `'='` : les deux utilisent le boutisme natif mais la taille et l'alignement du dernier sont standards.

The form `'!'` represents the network byte order which is always big-endian as defined in IETF RFC 1700.

Il n'y a pas de moyen de spécifier le boutisme contraire au boutisme natif (c'est-à-dire forcer la permutation des octets) ; utilisez le bon caractère entre `'<'` et `'>'`.

Notes :

(1) Le bourrage (*padding* en anglais) n'est automatiquement ajouté qu'entre les membres successifs de la structure. Il n'y a pas de bourrage au début ou à la fin de la structure agrégée.

(2) Il n'y a pas d'ajout de bourrage lorsque vous utilisez une taille et un alignement non-natifs, par exemple avec `'<'`, `'>'`, `'='` ou `'!'`.

(3) Pour aligner la fin d'une structure à l'alignement requis par un type particulier, terminez le format avec le code du type voulu et une valeur de répétition à zéro. Référez-vous à *Exemples*.

Caractères de format

Les caractères de format possèdent les significations suivantes ; la conversion entre les valeurs C et Python doit être évidente compte tenu des types concernés. La colonne « taille standard » fait référence à la taille en octets de la valeur agrégée avec l'utilisation de la taille standard (c'est-à-dire lorsque la chaîne de format commence par l'un des caractères suivants : `'<'`, `'>'`, `'!'` ou `'='`). Si vous utilisez la taille native, la taille de la valeur agrégée dépend de la plateforme.

Format	Type C	Type Python	Taille standard	Notes
`x`	octet de bourrage	pas de valeur		(7)
`c`	`char`	*bytes* (suite d'octets) de taille 1	1	
`b`	`signed char`	*int* (entier)	1	(1), (2)
`B`	`unsigned char`	*int* (entier)	1	(2)
`?`	`_Bool`	*bool* (booléen)	1	(1)
`h`	`short`	*int* (entier)	2	(2)
`H`	`unsigned short`	*int* (entier)	2	(2)
`i`	`int`	*int* (entier)	4	(2)
`I`	`unsigned int`	*int* (entier)	4	(2)
`l`	`long`	*int* (entier)	4	(2)
`L`	`unsigned long`	*int* (entier)	4	(2)
`q`	`long long`	*int* (entier)	8	(2)
`Q`	`unsigned long long`	*int* (entier)	8	(2)
`n`	`ssize_t`	*int* (entier)		(3)
`N`	`size_t`	*int* (entier)		(3)
`e`	(6)	*float* (nombre à virgule flottante)	2	(4)
`f`	`float`	*float* (nombre à virgule flottante)	4	(4)
`d`	`double`	*float* (nombre à virgule flottante)	8	(4)
`s`	`char[]`	*bytes* (séquence d'octets)		(9)
`p`	`char[]`	*bytes* (séquence d'octets)		(8)
`P`	`void*`	*int* (entier)		(5)

Additionally, if IEC 60559 compatible complex arithmetic (Annex G of the C11 standard) is supported, the following format characters are available :

Format	Type C	Type Python	Taille standard	Notes
`E`	`float complex`	complex	8	(10)
`C`	`double complex`	complex	16	(10)

Modifié dans la version 3.3 : ajouté la gestion des formats `'n'` et `'N'`.

Modifié dans la version 3.6 : ajouté la gestion du format `'e'`.

Modifié dans la version 3.14 : Added support for the `'E'` and `'C'` formats.

Notes :

(1) The `'?'` conversion code corresponds to the `_Bool` type defined by C standards since C99. In standard mode, it is represented by one byte.

(2) When attempting to pack a non-integer using any of the integer conversion codes, if the non-integer has a `__index__()` method then that method is called to convert the argument to an integer before packing.

Modifié dans la version 3.2 : Added use of the `__index__()` method for non-integers.

(3) Les codes de conversion `'n'` et `'N'` ne sont disponibles que pour la taille native (choisie par défaut ou à l'aide du caractère de boutisme `'@'`). Pour la taille standard, vous pouvez utiliser n'importe quel format d'entier qui convient à votre application.

(4) Pour les codes de conversion `'f'`, `'d'` et `'e'`, la représentation agrégée utilise respectivement le format IEEE 754 *binaire32*, *binaire64* ou *binaire16* (pour `'f'`, `'d'` ou `'e'` respectivement), quel que soit le format des nombres à virgule flottante de la plateforme.

(5) Le caractère de format `'P'` n'est disponible que pour le boutisme natif (choisi par défaut ou à l'aide du caractère `'@'` de boutisme). Le caractère de boutisme `'='` choisit d'utiliser un petit ou un gros en fonction du système hôte. Le module *struct* ne l'interprète pas comme un boutisme natif, donc le format `'P'` n'est pas disponible.

(6) Le type IEEE 754 *binaire16* « demie-précision » a été introduit en 2008 par la révision du standard IEEE 754. Il comprend un bit de signe, un exposant sur 5 bits et une précision de 11 bits (dont 10 bits sont explicitement stockés) ; il peut représenter les nombres entre environ `6.1e-05` et `6.5e+04` avec une précision

maximale. Ce type est rarement pris en charge par les compilateurs C : sur une machine courante, un *unsigned short* (entier court non signé) peut être utilisé pour le stockage mais pas pour les opérations mathématiques. Lisez la page Wikipédia (NdT : non traduite en français) half-precision floating-point format pour davantage d'informations.

(7) When packing, `'x'` inserts one NUL byte.

(8) Le caractère de format `'p'` sert à encoder une « chaîne Pascal », c'est-à-dire une courte chaîne de longueur variable, stockée dans un *nombre défini d'octets* dont la valeur est définie par la répétition. Le premier octet stocké est la longueur de la chaîne (dans la limite maximum de 255). Les octets composant la chaîne suivent. Si la chaîne passée à `pack()` est trop longue (supérieure à la valeur de la répétition moins 1), seuls les `count-1` premiers octets de la chaîne sont stockés. Si la chaîne est plus courte que `count-1`, des octets de bourrage nuls sont insérés de manière à avoir exactement *count* octets au final. Notez que pour `unpack()`, le caractère de format `'p'` consomme `count` octets mais que la chaîne renvoyée ne peut pas excéder 255 octets.

(9) For the `'s'` format character, the count is interpreted as the length of the bytes, not a repeat count like for the other format characters ; for example, `'10s'` means a single 10-byte string mapping to or from a single Python byte string, while `'10c'` means 10 separate one byte character elements (e.g., `cccccccccc`) mapping to or from ten different Python byte objects. (See *Exemples* for a concrete demonstration of the difference.) If a count is not given, it defaults to 1. For packing, the string is truncated or padded with null bytes as appropriate to make it fit. For unpacking, the resulting bytes object always has exactly the specified number of bytes. As a special case, `'0s'` means a single, empty string (while `'0c'` means 0 characters).

(10) For the `'E'` and `'C'` format characters, the packed representation uses the IEEE 754 binary32 and binary64 format for components of the complex number, regardless of the floating-point format used by the platform.

Un caractère de format peut être précédé par un entier indiquant le nombre de répétitions. Par exemple, la chaîne de format `'4h'` a exactement la même signification que `'hhhh'`.

Les caractères d'espacement entre les indications de format sont ignorés ; cependant, le nombre de répétitions et le format associé ne doivent pas être séparés par des caractères d'espacement.

Lors de l'agrégation d'une valeur `x` en utilisant l'un des formats pour les entiers (`'b'`, `'B'`, `'h'`, `'H'`, `'i'`, `'I'`, `'l'`, `'L'`, `'q'`, `'Q'`), si `x` est en dehors de l'intervalle du format spécifié, une `struct.error` est levée.

Modifié dans la version 3.1 : Previously, some of the integer formats wrapped out-of-range values and raised `DeprecationWarning` instead of `struct.error`.

Pour le caractère de format `'?'`, la valeur renvoyée est `True` ou `False`. Lors de l'agrégation, la valeur de vérité de l'objet argument est utilisée. La valeur agrégée est 0 ou 1 dans la représentation native ou standard et, lors de la dissociation, n'importe quelle valeur différente de zéro est renvoyée `True`.

Exemples

Note

Native byte order examples (designated by the `'@'` format prefix or lack of any prefix character) may not match what the reader's machine produces as that depends on the platform and compiler.

Pack and unpack integers of three different sizes, using big endian ordering :

```
>>> from struct import *
>>> pack(">bhl", 1, 2, 3)
b'\x01\x00\x02\x00\x00\x00\x03'
>>> unpack('>bhl', b'\x01\x00\x02\x00\x00\x00\x03')
(1, 2, 3)
>>> calcsize('>bhl')
7
```

Attempt to pack an integer which is too large for the defined field :

```
>>> pack(">h", 99999)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
struct.error: 'h' format requires -32768 <= number <= 32767
```

Demonstrate the difference between `'s'` and `'c'` format characters :

```
>>> pack("@ccc", b'1', b'2', b'3')
b'123'
>>> pack("@3s", b'123')
b'123'
```

Les champs dissociés peuvent être nommés en leur assignant des variables ou en encapsulant le résultat dans un *n*-uplet nommé :

```
>>> record = b'raymond   \x32\x12\x08\x01\x08'
>>> name, serialnum, school, gradelevel = unpack('<10sHHb', record)

>>> from collections import namedtuple
>>> Student = namedtuple('Student', 'name serialnum school gradelevel')
>>> Student._make(unpack('<10sHHb', record))
Student(name=b'raymond   ', serialnum=4658, school=264, gradelevel=8)
```

The ordering of format characters may have an impact on size in native mode since padding is implicit. In standard mode, the user is responsible for inserting any desired padding. Note in the first `pack` call below that three NUL bytes were added after the packed `'#'` to align the following integer on a four-byte boundary. In this example, the output was produced on a little endian machine :

```
>>> pack('@ci', b'#', 0x12131415)
b'#\x00\x00\x00\x15\x14\x13\x12'
>>> pack('@ic', 0x12131415, b'#')
b'\x15\x14\x13\x12#'
>>> calcsize('@ci')
8
>>> calcsize('@ic')
5
```

The following format `'llh0l'` results in two pad bytes being added at the end, assuming the platform's longs are aligned on 4-byte boundaries :

```
>>> pack('@llh0l', 1, 2, 3)
b'\x00\x00\x00\x01\x00\x00\x00\x02\x00\x03\x00\x00'
```

Voir aussi

Module ***array***
Stockage agrégé binaire de données homogènes.

Module ***json***
JSON encoder and decoder.

Module ***pickle***
Python object serialization.

7.1.3 Applications

Two main applications for the `struct` module exist, data interchange between Python and C code within an application or another application compiled using the same compiler (*native formats*), and data interchange between applications using agreed upon data layout (*standard formats*). Generally speaking, the format strings constructed for these two domains are distinct.

Native Formats

When constructing format strings which mimic native layouts, the compiler and machine architecture determine byte ordering and padding. In such cases, the `@` format character should be used to specify native byte ordering and data sizes. Internal pad bytes are normally inserted automatically. It is possible that a zero-repeat format code will be needed at the end of a format string to round up to the correct byte boundary for proper alignment of consecutive chunks of data.

Consider these two simple examples (on a 64-bit, little-endian machine) :

```
>>> calcsize('@lhl')
24
>>> calcsize('@llh')
18
```

Data is not padded to an 8-byte boundary at the end of the second format string without the use of extra padding. A zero-repeat format code solves that problem :

```
>>> calcsize('@llh0l')
24
```

The `'x'` format code can be used to specify the repeat, but for native formats it is better to use a zero-repeat format like `'0l'`.

By default, native byte ordering and alignment is used, but it is better to be explicit and use the `'@'` prefix character.

Standard Formats

When exchanging data beyond your process such as networking or storage, be precise. Specify the exact byte order, size, and alignment. Do not assume they match the native order of a particular machine. For example, network byte order is big-endian, while many popular CPUs are little-endian. By defining this explicitly, the user need not care about the specifics of the platform their code is running on. The first character should typically be `<` or `>` (or `!`). Padding is the responsibility of the programmer. The zero-repeat format character won't work. Instead, the user must explicitly add `'x'` pad bytes where needed. Revisiting the examples from the previous section, we have :

```
>>> calcsize('<qh6xq')
24
>>> pack('<qh6xq', 1, 2, 3) == pack('@lhl', 1, 2, 3)
True
>>> calcsize('@llh')
18
>>> pack('@llh', 1, 2, 3) == pack('<qqh', 1, 2, 3)
True
>>> calcsize('<qqh6x')
24
>>> calcsize('@llh0l')
24
>>> pack('@llh0l', 1, 2, 3) == pack('<qqh6x', 1, 2, 3)
True
```

The above results (executed on a 64-bit machine) aren't guaranteed to match when executed on different machines. For example, the examples below were executed on a 32-bit machine :

```
>>> calcsize('<qqh6x')
24
>>> calcsize('@llh0l')
12
>>> pack('@llh0l', 1, 2, 3) == pack('<qqh6x', 1, 2, 3)
False
```

7.1.4 Classes

Le module `struct` définit aussi le type suivant :

class struct.**Struct**(*format*)

Return a new Struct object which writes and reads binary data according to the format string *format*. Creating a `Struct` object once and calling its methods is more efficient than calling module-level functions with the same format since the format string is only compiled once.

> **Note**
>
> The compiled versions of the most recent format strings passed to the module-level functions are cached, so programs that use only a few format strings needn't worry about reusing a single `Struct` instance.

Les objets `Struct` compilés gèrent les méthodes et attributs suivants :

pack(*v1*, *v2*, ...)

Identique à la fonction `pack()`, en utilisant le format compilé (`len(result)` vaut `size`).

pack_into(*buffer*, *offset*, *v1*, *v2*, ...)

Identique à la fonction `pack_into()`, en utilisant le format compilé.

unpack(*buffer*)

Identique à la fonction `unpack()`, en utilisant le format compilé. La taille du tampon *buffer* en octets doit valoir `size`.

unpack_from(*buffer*, *offset=0*)

Identique à la fonction `unpack_from()`, en utilisant le format compilé. La taille du tampon *buffer* en octets, en commençant à la position *offset*, doit valoir au moins `size`.

iter_unpack(*buffer*)

Identique à la fonction `iter_unpack()`, en utilisant le format compilé. La taille du tampon *buffer* en octets doit être un multiple de `size`.

Ajouté dans la version 3.4.

format

La chaîne de format utilisée pour construire l'objet `Struct`.

Modifié dans la version 3.7 : la chaîne de format est maintenant de type `str` au lieu de `bytes`.

size

La taille calculée de la structure agrégée (et donc de l'objet `bytes` produit par la méthode `pack()`) correspondante à `format`.

Modifié dans la version 3.13 : The *repr()* of structs has changed. It is now :

```
>>> Struct('i')
Struct('i')
```

7.2 codecs --- Codec registry and base classes

Code source : Lib/codecs.py

This module defines base classes for standard Python codecs (encoders and decoders) and provides access to the internal Python codec registry, which manages the codec and error handling lookup process. Most standard codecs are *text encodings*, which encode text to bytes (and decode bytes to text), but there are also codecs provided that encode text to text, and bytes to bytes. Custom codecs may encode and decode between arbitrary types, but some module features are restricted to be used specifically with *text encodings* or with codecs that encode to `bytes`.

Le module définit les fonctions suivantes pour encoder et décoder à l'aide de n'importe quel codec :

`codecs.encode`(*obj*, *encoding='utf-8'*, *errors='strict'*)

Encode *obj* en utilisant le codec enregistré pour *encoding*.

Vous pouvez spécifier *errors* pour définir la façon de gérer les erreurs. Le gestionnaire d'erreurs par défaut est `'strict'`, ce qui veut dire qu'une erreur lors de l'encodage lève `ValueError` (ou une sous-classe spécifique du codec, telle que `UnicodeEncodeError`). Référez-vous aux *classes de base des codecs* pour plus d'informations sur la gestion des erreurs par les codecs.

`codecs.decode`(*obj*, *encoding='utf-8'*, *errors='strict'*)

Décode *obj* en utilisant le codec enregistré pour *encoding*.

Vous pouvez spécifier *errors* pour définir la façon de gérer les erreurs. Le gestionnaire d'erreurs par défaut est `'strict'`, ce qui veut dire qu'une erreur lors du décodage lève `ValueError` (ou une sous-classe spécifique du codec, telle que `UnicodeDecodeError`). Référez-vous aux *classes de base des codecs* pour plus d'informations sur la gestion des erreurs par les codecs.

Les détails complets de chaque codec peuvent être examinés directement :

`codecs.lookup`(*encoding*)

Recherche les informations relatives au codec dans le registre des codecs de Python et renvoie l'objet `CodecInfo` tel que défini ci-dessous.

Les encodeurs sont recherchés en priorité dans le cache du registre. S'ils n'y sont pas, la liste des fonctions de recherche enregistrées est passée en revue. Si aucun objet `CodecInfo` n'est trouvé, une `LookupError` est levée. Sinon, l'objet `CodecInfo` est mis en cache et renvoyé vers l'appelant.

class `codecs.CodecInfo`(*encode*, *decode*, *streamreader=None*, *streamwriter=None*, *incrementalencoder=None*, *incrementaldecoder=None*, *name=None*)

Les détails d'un codec trouvé dans le registre des codecs. Les arguments du constructeur sont stockés dans les attributs éponymes :

name

Le nom de l'encodeur.

encode

decode

Les fonctions d'encodage et de décodage. Ces fonctions ou méthodes doivent avoir la même interface que les méthodes `encode()` et `decode()` des instances de Codec (voir *Interface des codecs*). Les fonctions et méthodes sont censées fonctionner sans état interne.

incrementalencoder

incrementaldecoder

Classes d'encodeurs et de décodeurs incrémentaux ou fonctions usines. Elles doivent avoir respectivement les mêmes interfaces que celles définies par les classes de base `IncrementalEncoder` et `IncrementalDecoder`. Les codecs incrémentaux peuvent conserver des états internes.

streamwriter

streamreader

Classes d'écriture et de lecture de flux ou fonctions usines. Elles doivent avoir les mêmes interfaces que celles définies par les classes de base `StreamWriter` et `StreamReader`, respectivement. Les codecs de flux peuvent conserver un état interne.

Pour simplifier l'accès aux différents composants du codec, le module fournit les fonctions supplémentaires suivantes qui utilisent `lookup()` pour la recherche du codec :

codecs.**getencoder**(*encoding*)

Recherche le codec pour l'encodage *encoding* et renvoie sa fonction d'encodage.

Lève une `LookupError` si l'encodage *encoding* n'est pas trouvé.

codecs.**getdecoder**(*encoding*)

Recherche le codec pour l'encodage *encoding* et renvoie sa fonction de décodage.

Lève une `LookupError` si l'encodage *encoding* n'est pas trouvé.

codecs.**getincrementalencoder**(*encoding*)

Recherche le codec pour l'encodage *encoding* et renvoie sa classe d'encodage incrémental ou la fonction usine.

Lève une `LookupError` si l'encodage *encoding* n'est pas trouvé ou si le codec ne gère pas l'encodage incrémental.

codecs.**getincrementaldecoder**(*encoding*)

Recherche le codec pour l'encodage *encoding* et renvoie sa classe de décodage incrémental ou la fonction usine.

Lève une `LookupError` si l'encodage *encoding* n'est pas trouvé ou si le codec ne gère pas le décodage incrémental.

codecs.**getreader**(*encoding*)

Recherche le codec pour l'encodage *encoding* et renvoie sa classe `StreamReader` ou la fonction usine.

Lève une `LookupError` si l'encodage *encoding* n'est pas trouvé.

codecs.**getwriter**(*encoding*)

Recherche le codec pour l'encodage *encoding* et renvoie sa classe `StreamWriter` ou la fonction usine.

Lève une `LookupError` si l'encodage *encoding* n'est pas trouvé.

Les codecs personnalisés sont mis à disposition en enregistrant une fonction de recherche de codecs adaptée :

codecs.**register**(*search_function*)

Register a codec search function. Search functions are expected to take one argument, being the encoding name in all lower case letters with hyphens and spaces converted to underscores, and return a `CodecInfo` object. In case a search function cannot find a given encoding, it should return `None`.

Modifié dans la version 3.9 : Hyphens and spaces are converted to underscore.

codecs.**unregister**(*search_function*)

Unregister a codec search function and clear the registry's cache. If the search function is not registered, do nothing.

Ajouté dans la version 3.10.

Alors qu'il est recommandé d'utiliser la fonction native `open()` et le module associé `io` pour travailler avec des fichiers texte encodés, le présent module fournit des fonctions et classes utilitaires supplémentaires qui permettent l'utilisation d'une plus large gamme de codecs si vous travaillez avec des fichiers binaires :

codecs.**open**(*filename*, *mode='r'*, *encoding=None*, *errors='strict'*, *buffering=-1*)

Ouvre un fichier encodé en utilisant le *mode* donné et renvoie une instance de `StreamReaderWriter`, permettant un encodage-décodage transparent. Le mode de fichier par défaut est `'r'`, ce qui signifie que le fichier est ouvert en lecture.

> **Note**
>
> If *encoding* is not `None`, then the underlying encoded files are always opened in binary mode. No automatic conversion of `'\n'` is done on reading and writing. The *mode* argument may be any binary mode acceptable to the built-in `open()` function ; the `'b'` is automatically added.

encoding spécifie l'encodage à utiliser pour le fichier. Tout encodage qui encode et décode des octets (type *bytes*) est autorisé et les types de données pris en charge par les méthodes relatives aux fichiers dépendent du codec utilisé.

errors peut être spécifié pour définir la gestion des erreurs. La valeur par défaut est `'strict'`, ce qui lève une `ValueError` en cas d'erreur lors du codage.

buffering a la même signification que pour la fonction native `open()`. Il vaut `-1` par défaut, ce qui signifie que la taille par défaut du tampon est utilisée.

Modifié dans la version 3.11 : le mode `'U` a été enlevé.

`codecs.`**`EncodedFile`**(*file*, *data_encoding*, *file_encoding=None*, *errors='strict'*)

Renvoie une instance de `StreamRecoder`, version encapsulée de *file* qui fournit un transcodage transparent. Le fichier original est fermé quand la version encapsulée est fermée.

Les données écrites dans un fichier encapsulant sont décodées en fonction du *data_encoding* spécifié puis écrites vers le fichier original en tant que *bytes* en utilisant *file_encoding*. Les octets lus dans le fichier original sont décodés conformément à *file_encoding* et le résultat est encodé en utilisant *data_encoding*.

Si *file_encoding* n'est pas spécifié, la valeur par défaut est *data_encoding*.

errors peut être spécifié pour définir la gestion des erreurs. La valeur par défaut est `'strict'`, ce qui lève une `ValueError` en cas d'erreur lors du codage.

`codecs.`**`iterencode`**(*iterator*, *encoding*, *errors='strict'*, ***kwargs*)

Utilise un encodeur incrémental pour encoder de manière itérative l'entrée fournie par *iterator*. Cette fonction est un *générateur*. L'argument *errors* (ainsi que tout autre argument passé par son nom) est transmis à l'encodeur incrémental.

Cette fonction nécessite que le codec accepte les objets texte (classe `str`) en entrée. Par conséquent, il ne prend pas en charge les encodeurs *bytes* vers *bytes* tels que `base64_codec`.

`codecs.`**`iterdecode`**(*iterator*, *encoding*, *errors='strict'*, ***kwargs*)

Utilise un décodeur incrémental pour décoder de manière itérative l'entrée fournie par *iterator*. Cette fonction est un *générateur*. L'argument *errors* (ainsi que tout autre argument passé par son nom) est transmis au décodeur incrémental.

Cette fonction requiert que le codec accepte les objets `bytes` en entrée. Par conséquent, elle ne prend pas en charge les encodeurs de texte vers texte tels que `rot_13`, bien que `rot_13` puisse être utilisé de manière équivalente avec `iterencode()`.

Le module fournit également les constantes suivantes qui sont utiles pour lire et écrire les fichiers dépendants de la plateforme :

`codecs.`**`BOM`**

`codecs.`**`BOM_BE`**

`codecs.`**`BOM_LE`**

`codecs.`**`BOM_UTF8`**

`codecs.`**`BOM_UTF16`**

`codecs.`**`BOM_UTF16_BE`**

`codecs.`**`BOM_UTF16_LE`**

`codecs.`**`BOM_UTF32`**

`codecs.`**`BOM_UTF32_BE`**

codecs.**BOM_UTF32_LE**

Ces constantes définissent diverses séquences d'octets, les marques d'ordre d'octets (BOM pour *byte order mark* en anglais) Unicode pour plusieurs encodages. Elles sont utilisées dans les flux de données UTF-16 et UTF-32 pour indiquer l'ordre des octets utilisé, et dans UTF-8 comme signature Unicode. *BOM_UTF16* vaut soit *BOM_UTF16_BE*, soit *BOM_UTF16_LE* selon le boutisme natif de la plateforme, *BOM* est un alias pour *BOM_UTF16*, *BOM_LE* pour *BOM_UTF16_LE* et *BOM_BE* pour *BOM_UTF16_BE*. Les autres sont les marques BOM dans les encodages UTF-8 et UTF-32.

7.2.1 Classes de base de codecs

Le module *codecs* définit un ensemble de classes de base qui spécifient les interfaces pour travailler avec des objets codecs et qui peuvent également être utilisées comme base pour des implémentations de codecs personnalisés.

Chaque codec doit définir quatre interfaces pour être utilisable comme codec en Python : codeur sans état, décodeur sans état, lecteur de flux et écrivain de flux. Le lecteur et l'écrivain de flux réutilisent généralement l'encodeur-décodeur sans état pour implémenter les protocoles de fichiers. Les auteurs de codecs doivent également définir comment le codec gère les erreurs d'encodage et de décodage.

Gestionnaires d'erreurs

To simplify and standardize error handling, codecs may implement different error handling schemes by accepting the *errors* string argument :

```
>>> 'German ß, ♫'.encode(encoding='ascii', errors='backslashreplace')
b'German \\xdf, \\u266c'
>>> 'German ß, ♫'.encode(encoding='ascii', errors='xmlcharrefreplace')
b'German &#223;, &#9836;'
```

The following error handlers can be used with all Python *Standard Encodings* codecs :

Valeur	Signification
`'strict'`	Raise *UnicodeError* (or a subclass), this is the default. Implemented in *strict_errors()*.
`'ignore'`	Ignore les données incorrectement formatées et continue sans rien signaler. Implémenté dans *ignore_errors()*.
`'replace'`	Replace with a replacement marker. On encoding, use `?` (ASCII character). On decoding, use `\ufffd` (U+FFFD, the official REPLACEMENT CHARACTER). Implemented in *replace_errors()*.
`'backslashreplace'`	Replace with backslashed escape sequences. On encoding, use hexadecimal form of Unicode code point with formats `\xhh` `\uxxxx` `\Uxxxxxxxx`. On decoding, use hexadecimal form of byte value with format `\xhh`. Implemented in *backslashreplace_errors()*.
`'surrogateescape'`	Lors du décodage, remplace un octet par un code de substitution individuel allant de `U+DC80` à `U+DCFF`. Ce code est reconverti vers l'octet de départ quand le gestionnaire d'erreurs `'surrogateescape'` est utilisé pour l'encodage des données (voir la **PEP 383** pour plus de détails).

The following error handlers are only applicable to encoding (within *text encodings*) :

Valeur	Signification
`'xmlcharref`	Replace with XML/HTML numeric character reference, which is a decimal form of Unicode code point with format `&#num;`. Implemented in *xmlcharrefreplace_errors()*.
`'namereplac`	Replace with `\N{...}` escape sequences, what appears in the braces is the Name property from Unicode Character Database. Implemented in *namereplace_errors()*.

En plus, le gestionnaire d'erreurs suivant est spécifique aux codecs suivants :

Valeur	Codecs	Signification
`'surrog`	utf-8, utf-16, utf-32, utf-16-be, utf-16-le, utf-32-be, utf-32-le	Allow encoding and decoding surrogate code point (`U+D800` - `U+DFFF`) as normal code point. Otherwise these codecs treat the presence of surrogate code point in `str` as an error.

Ajouté dans la version 3.1 : les gestionnaires d'erreurs `'surrogateescape'` et `'surrogatepass'`.

Modifié dans la version 3.4 : The `'surrogatepass'` error handler now works with utf-16* and utf-32* codecs.

Ajouté dans la version 3.5 : le gestionnaire d'erreurs `'namereplace'`.

Modifié dans la version 3.5 : The `'backslashreplace'` error handler now works with decoding and translating.

L'ensemble des valeurs autorisées peut être étendu en enregistrant un nouveau gestionnaire d'erreurs nommé :

`codecs.`**`register_error`**(*name*, *error_handler*)

Register the error handling function *error_handler* under the name *name*. The *error_handler* argument will be called during encoding and decoding in case of an error, when *name* is specified as the errors parameter.

For encoding, *error_handler* will be called with a `UnicodeEncodeError` instance, which contains information about the location of the error. The error handler must either raise this or a different exception, or return a tuple with a replacement for the unencodable part of the input and a position where encoding should continue. The replacement may be either `str` or `bytes`. If the replacement is bytes, the encoder will simply copy them into the output buffer. If the replacement is a string, the encoder will encode the replacement. Encoding continues on original input at the specified position. Negative position values will be treated as being relative to the end of the input string. If the resulting position is out of bound an `IndexError` will be raised.

Decoding and translating works similarly, except `UnicodeDecodeError` or `UnicodeTranslateError` will be passed to the handler and that the replacement from the error handler will be put into the output directly.

Previously registered error handlers (including the standard error handlers) can be looked up by name :

`codecs.`**`lookup_error`**(*name*)

Return the error handler previously registered under the name *name*.

Raises a `LookupError` in case the handler cannot be found.

The following standard error handlers are also made available as module level functions :

`codecs.`**`strict_errors`**(*exception*)

Implements the `'strict'` error handling.

Each encoding or decoding error raises a `UnicodeError`.

`codecs.`**`ignore_errors`**(*exception*)

Implements the `'ignore'` error handling.

Malformed data is ignored ; encoding or decoding is continued without further notice.

`codecs.`**`replace_errors`**(*exception*)

Implements the `'replace'` error handling.

Substitutes `?` (ASCII character) for encoding errors or `⁇` (U+FFFD, the official REPLACEMENT CHARACTER) for decoding errors.

`codecs.`**`backslashreplace_errors`**(*exception*)

Implements the `'backslashreplace'` error handling.

Malformed data is replaced by a backslashed escape sequence. On encoding, use the hexadecimal form of Unicode code point with formats `\xhh` `\uxxxx` `\Uxxxxxxxx`. On decoding, use the hexadecimal form of byte value with format `\xhh`.

Modifié dans la version 3.5 : Works with decoding and translating.

codecs.**xmlcharrefreplace_errors**(*exception*)

Implements the `'xmlcharrefreplace'` error handling (for encoding within *text encoding* only).

The unencodable character is replaced by an appropriate XML/HTML numeric character reference, which is a decimal form of Unicode code point with format `&#num;` .

codecs.**namereplace_errors**(*exception*)

Implements the `'namereplace'` error handling (for encoding within *text encoding* only).

The unencodable character is replaced by a `\N{...}` escape sequence. The set of characters that appear in the braces is the Name property from Unicode Character Database. For example, the German lowercase letter `'ß'` will be converted to byte sequence `\N{LATIN SMALL LETTER SHARP S}` .

Ajouté dans la version 3.5.

Stateless Encoding and Decoding

The base `Codec` class defines these methods which also define the function interfaces of the stateless encoder and decoder :

class codecs.**Codec**

encode(*input*, *errors='strict'*)

Encodes the object *input* and returns a tuple (output object, length consumed). For instance, *text encoding* converts a string object to a bytes object using a particular character set encoding (e.g., `cp1252` or `iso-8859-1`).

The *errors* argument defines the error handling to apply. It defaults to `'strict'` handling.

The method may not store state in the `Codec` instance. Use `StreamWriter` for codecs which have to keep state in order to make encoding efficient.

The encoder must be able to handle zero length input and return an empty object of the output object type in this situation.

decode(*input*, *errors='strict'*)

Decodes the object *input* and returns a tuple (output object, length consumed). For instance, for a *text encoding*, decoding converts a bytes object encoded using a particular character set encoding to a string object.

For text encodings and bytes-to-bytes codecs, *input* must be a bytes object or one which provides the read-only buffer interface -- for example, buffer objects and memory mapped files.

The *errors* argument defines the error handling to apply. It defaults to `'strict'` handling.

The method may not store state in the `Codec` instance. Use `StreamReader` for codecs which have to keep state in order to make decoding efficient.

The decoder must be able to handle zero length input and return an empty object of the output object type in this situation.

Incremental Encoding and Decoding

The `IncrementalEncoder` and `IncrementalDecoder` classes provide the basic interface for incremental encoding and decoding. Encoding/decoding the input isn't done with one call to the stateless encoder/decoder function, but with multiple calls to the `encode()`/`decode()` method of the incremental encoder/decoder. The incremental encoder/decoder keeps track of the encoding/decoding process during method calls.

The joined output of calls to the `encode()`/`decode()` method is the same as if all the single inputs were joined into one, and this input was encoded/decoded with the stateless encoder/decoder.

IncrementalEncoder Objects

The `IncrementalEncoder` class is used for encoding an input in multiple steps. It defines the following methods which every incremental encoder must define in order to be compatible with the Python codec registry.

class codecs.**IncrementalEncoder**(*errors='strict'*)

Constructor for an `IncrementalEncoder` instance.

All incremental encoders must provide this constructor interface. They are free to add additional keyword arguments, but only the ones defined here are used by the Python codec registry.

The `IncrementalEncoder` may implement different error handling schemes by providing the *errors* keyword argument. See *Gestionnaires d'erreurs* for possible values.

The *errors* argument will be assigned to an attribute of the same name. Assigning to this attribute makes it possible to switch between different error handling strategies during the lifetime of the `IncrementalEncoder` object.

encode(*object*, *final=False*)

Encodes *object* (taking the current state of the encoder into account) and returns the resulting encoded object. If this is the last call to `encode()` *final* must be true (the default is false).

reset()

Reset the encoder to the initial state. The output is discarded : call `.encode(object, final=True)`, passing an empty byte or text string if necessary, to reset the encoder and to get the output.

getstate()

Return the current state of the encoder which must be an integer. The implementation should make sure that `0` is the most common state. (States that are more complicated than integers can be converted into an integer by marshaling/pickling the state and encoding the bytes of the resulting string into an integer.)

setstate(*state*)

Set the state of the encoder to *state*. *state* must be an encoder state returned by `getstate()`.

IncrementalDecoder Objects

The `IncrementalDecoder` class is used for decoding an input in multiple steps. It defines the following methods which every incremental decoder must define in order to be compatible with the Python codec registry.

class codecs.**IncrementalDecoder**(*errors='strict'*)

Constructor for an `IncrementalDecoder` instance.

All incremental decoders must provide this constructor interface. They are free to add additional keyword arguments, but only the ones defined here are used by the Python codec registry.

The `IncrementalDecoder` may implement different error handling schemes by providing the *errors* keyword argument. See *Gestionnaires d'erreurs* for possible values.

The *errors* argument will be assigned to an attribute of the same name. Assigning to this attribute makes it possible to switch between different error handling strategies during the lifetime of the `IncrementalDecoder` object.

decode(*object*, *final=False*)

Decodes *object* (taking the current state of the decoder into account) and returns the resulting decoded object. If this is the last call to `decode()` *final* must be true (the default is false). If *final* is true the decoder must decode the input completely and must flush all buffers. If this isn't possible (e.g. because of incomplete byte sequences at the end of the input) it must initiate error handling just like in the stateless case (which might raise an exception).

reset()

Reset the decoder to the initial state.

getstate()

Return the current state of the decoder. This must be a tuple with two items, the first must be the buffer

containing the still undecoded input. The second must be an integer and can be additional state info. (The implementation should make sure that `0` is the most common additional state info.) If this additional state info is `0` it must be possible to set the decoder to the state which has no input buffered and `0` as the additional state info, so that feeding the previously buffered input to the decoder returns it to the previous state without producing any output. (Additional state info that is more complicated than integers can be converted into an integer by marshaling/pickling the info and encoding the bytes of the resulting string into an integer.)

setstate(*state*)

Set the state of the decoder to *state*. *state* must be a decoder state returned by `getstate()`.

Stream Encoding and Decoding

The `StreamWriter` and `StreamReader` classes provide generic working interfaces which can be used to implement new encoding submodules very easily. See `encodings.utf_8` for an example of how this is done.

StreamWriter Objects

The `StreamWriter` class is a subclass of `Codec` and defines the following methods which every stream writer must define in order to be compatible with the Python codec registry.

class codecs.**StreamWriter**(*stream*, *errors='strict'*)

Constructor for a `StreamWriter` instance.

All stream writers must provide this constructor interface. They are free to add additional keyword arguments, but only the ones defined here are used by the Python codec registry.

The *stream* argument must be a file-like object open for writing text or binary data, as appropriate for the specific codec.

The `StreamWriter` may implement different error handling schemes by providing the *errors* keyword argument. See *Gestionnaires d'erreurs* for the standard error handlers the underlying stream codec may support.

The *errors* argument will be assigned to an attribute of the same name. Assigning to this attribute makes it possible to switch between different error handling strategies during the lifetime of the `StreamWriter` object.

write(*object*)

Writes the object's contents encoded to the stream.

writelines(*list*)

Writes the concatenated iterable of strings to the stream (possibly by reusing the `write()` method). Infinite or very large iterables are not supported. The standard bytes-to-bytes codecs do not support this method.

reset()

Resets the codec buffers used for keeping internal state.

Calling this method should ensure that the data on the output is put into a clean state that allows appending of new fresh data without having to rescan the whole stream to recover state.

In addition to the above methods, the `StreamWriter` must also inherit all other methods and attributes from the underlying stream.

StreamReader Objects

The `StreamReader` class is a subclass of `Codec` and defines the following methods which every stream reader must define in order to be compatible with the Python codec registry.

class codecs.**StreamReader**(*stream*, *errors='strict'*)

Constructor for a `StreamReader` instance.

All stream readers must provide this constructor interface. They are free to add additional keyword arguments, but only the ones defined here are used by the Python codec registry.

The *stream* argument must be a file-like object open for reading text or binary data, as appropriate for the specific codec.

The `StreamReader` may implement different error handling schemes by providing the *errors* keyword argument. See *Gestionnaires d'erreurs* for the standard error handlers the underlying stream codec may support.

The *errors* argument will be assigned to an attribute of the same name. Assigning to this attribute makes it possible to switch between different error handling strategies during the lifetime of the `StreamReader` object.

The set of allowed values for the *errors* argument can be extended with `register_error()`.

read(*size=-1*, *chars=-1*, *firstline=False*)

Decodes data from the stream and returns the resulting object.

The *chars* argument indicates the number of decoded code points or bytes to return. The `read()` method will never return more data than requested, but it might return less, if there is not enough available.

The *size* argument indicates the approximate maximum number of encoded bytes or code points to read for decoding. The decoder can modify this setting as appropriate. The default value -1 indicates to read and decode as much as possible. This parameter is intended to prevent having to decode huge files in one step.

The *firstline* flag indicates that it would be sufficient to only return the first line, if there are decoding errors on later lines.

The method should use a greedy read strategy meaning that it should read as much data as is allowed within the definition of the encoding and the given size, e.g. if optional encoding endings or state markers are available on the stream, these should be read too.

readline(*size=None*, *keepends=True*)

Read one line from the input stream and return the decoded data.

size, if given, is passed as size argument to the stream's `read()` method.

If *keepends* is false line-endings will be stripped from the lines returned.

readlines(*sizehint=None*, *keepends=True*)

Read all lines available on the input stream and return them as a list of lines.

Line-endings are implemented using the codec's `decode()` method and are included in the list entries if *keepends* is true.

sizehint, if given, is passed as the *size* argument to the stream's `read()` method.

reset()

Resets the codec buffers used for keeping internal state.

Note that no stream repositioning should take place. This method is primarily intended to be able to recover from decoding errors.

In addition to the above methods, the `StreamReader` must also inherit all other methods and attributes from the underlying stream.

StreamReaderWriter Objects

The `StreamReaderWriter` is a convenience class that allows wrapping streams which work in both read and write modes.

The design is such that one can use the factory functions returned by the `lookup()` function to construct the instance.

class codecs.**StreamReaderWriter**(*stream*, *Reader*, *Writer*, *errors='strict'*)

Creates a `StreamReaderWriter` instance. *stream* must be a file-like object. *Reader* and *Writer* must be factory functions or classes providing the `StreamReader` and `StreamWriter` interface resp. Error handling is done in the same way as defined for the stream readers and writers.

`StreamReaderWriter` instances define the combined interfaces of `StreamReader` and `StreamWriter` classes. They inherit all other methods and attributes from the underlying stream.

StreamRecoder Objects

The `StreamRecoder` translates data from one encoding to another, which is sometimes useful when dealing with different encoding environments.

The design is such that one can use the factory functions returned by the `lookup()` function to construct the instance.

class codecs.**StreamRecoder**(*stream*, *encode*, *decode*, *Reader*, *Writer*, *errors='strict'*)

Creates a `StreamRecoder` instance which implements a two-way conversion : *encode* and *decode* work on the frontend — the data visible to code calling `read()` and `write()`, while *Reader* and *Writer* work on the backend — the data in *stream*.

You can use these objects to do transparent transcodings, e.g., from Latin-1 to UTF-8 and back.

The *stream* argument must be a file-like object.

The *encode* and *decode* arguments must adhere to the `Codec` interface. *Reader* and *Writer* must be factory functions or classes providing objects of the `StreamReader` and `StreamWriter` interface respectively.

Error handling is done in the same way as defined for the stream readers and writers.

`StreamRecoder` instances define the combined interfaces of `StreamReader` and `StreamWriter` classes. They inherit all other methods and attributes from the underlying stream.

7.2.2 Encodings and Unicode

Strings are stored internally as sequences of code points in range `U+0000`--`U+10FFFF`. (See **PEP 393** for more details about the implementation.) Once a string object is used outside of CPU and memory, endianness and how these arrays are stored as bytes become an issue. As with other codecs, serialising a string into a sequence of bytes is known as *encoding*, and recreating the string from the sequence of bytes is known as *decoding*. There are a variety of different text serialisation codecs, which are collectivity referred to as *text encodings*.

The simplest text encoding (called `'latin-1'` or `'iso-8859-1'`) maps the code points 0--255 to the bytes `0x0`--`0xff`, which means that a string object that contains code points above `U+00FF` can't be encoded with this codec. Doing so will raise a `UnicodeEncodeError` that looks like the following (although the details of the error message may differ) : `UnicodeEncodeError: 'latin-1' codec can't encode character '\u1234' in position 3: ordinal not in range(256).`

There's another group of encodings (the so called charmap encodings) that choose a different subset of all Unicode code points and how these code points are mapped to the bytes `0x0`--`0xff`. To see how this is done simply open e.g. `encodings/cp1252.py` (which is an encoding that is used primarily on Windows). There's a string constant with 256 characters that shows you which character is mapped to which byte value.

All of these encodings can only encode 256 of the 1114112 code points defined in Unicode. A simple and straightforward way that can store each Unicode code point, is to store each code point as four consecutive bytes. There are two possibilities : store the bytes in big endian or in little endian order. These two encodings are called `UTF-32-BE` and `UTF-32-LE` respectively. Their disadvantage is that if e.g. you use `UTF-32-BE` on a little endian machine you will always have to swap bytes on encoding and decoding. `UTF-32` avoids this problem : bytes will always be in natural endianness. When these bytes are read by a CPU with a different endianness, then bytes have to be swapped though. To be able to detect the endianness of a `UTF-16` or `UTF-32` byte sequence, there's the so called BOM ("Byte Order Mark"). This is the Unicode character `U+FEFF`. This character can be prepended to every `UTF-16` or `UTF-32` byte sequence. The byte swapped version of this character (`0xFFFE`) is an illegal character that may not appear in a Unicode text. So when the first character in a `UTF-16` or `UTF-32` byte sequence appears to be a `U+FFFE` the bytes have to be swapped on decoding. Unfortunately the character `U+FEFF` had a second purpose as a `ZERO WIDTH NO-BREAK SPACE` : a character that has no width and doesn't allow a word to be split. It can e.g. be used to give hints to a ligature algorithm. With Unicode 4.0 using `U+FEFF` as a `ZERO WIDTH NO-BREAK SPACE` has been deprecated (with `U+2060` (`WORD JOINER`) assuming this role). Nevertheless Unicode software still must be able to handle `U+FEFF` in both roles : as a BOM it's a device to determine the storage layout of the encoded bytes, and

vanishes once the byte sequence has been decoded into a string ; as a `ZERO WIDTH NO-BREAK SPACE` it's a normal character that will be decoded like any other.

There's another encoding that is able to encode the full range of Unicode characters : UTF-8. UTF-8 is an 8-bit encoding, which means there are no issues with byte order in UTF-8. Each byte in a UTF-8 byte sequence consists of two parts : marker bits (the most significant bits) and payload bits. The marker bits are a sequence of zero to four `1` bits followed by a `0` bit. Unicode characters are encoded like this (with x being payload bits, which when concatenated give the Unicode character) :

Range	Encoding
`U-00000000` ... `U-0000007F`	0xxxxxxx
`U-00000080` ... `U-000007FF`	110xxxxx 10xxxxxx
`U-00000800` ... `U-0000FFFF`	1110xxxx 10xxxxxx 10xxxxxx
`U-00010000` ... `U-0010FFFF`	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

The least significant bit of the Unicode character is the rightmost x bit.

As UTF-8 is an 8-bit encoding no BOM is required and any `U+FEFF` character in the decoded string (even if it's the first character) is treated as a `ZERO WIDTH NO-BREAK SPACE`.

Without external information it's impossible to reliably determine which encoding was used for encoding a string. Each charmap encoding can decode any random byte sequence. However that's not possible with UTF-8, as UTF-8 byte sequences have a structure that doesn't allow arbitrary byte sequences. To increase the reliability with which a UTF-8 encoding can be detected, Microsoft invented a variant of UTF-8 (that Python calls `"utf-8-sig"`) for its Notepad program : Before any of the Unicode characters is written to the file, a UTF-8 encoded BOM (which looks like this as a byte sequence : `0xef`, `0xbb`, `0xbf`) is written. As it's rather improbable that any charmap encoded file starts with these byte values (which would e.g. map to

> LATIN SMALL LETTER I WITH DIAERESIS
> RIGHT-POINTING DOUBLE ANGLE QUOTATION MARK
> INVERTED QUESTION MARK

in iso-8859-1), this increases the probability that a `utf-8-sig` encoding can be correctly guessed from the byte sequence. So here the BOM is not used to be able to determine the byte order used for generating the byte sequence, but as a signature that helps in guessing the encoding. On encoding the utf-8-sig codec will write `0xef`, `0xbb`, `0xbf` as the first three bytes to the file. On decoding `utf-8-sig` will skip those three bytes if they appear as the first three bytes in the file. In UTF-8, the use of the BOM is discouraged and should generally be avoided.

7.2.3 Standard Encodings

Python comes with a number of codecs built-in, either implemented as C functions or with dictionaries as mapping tables. The following table lists the codecs by name, together with a few common aliases, and the languages for which the encoding is likely used. Neither the list of aliases nor the list of languages is meant to be exhaustive. Notice that spelling alternatives that only differ in case or use a hyphen instead of an underscore are also valid aliases ; therefore, e.g. `'utf-8'` is a valid alias for the `'utf_8'` codec.

On Windows, `cpXXX` codecs are available for all code pages. But only codecs listed in the following table are guarantead to exist on other platforms.

Particularité de l'implémentation CPython : Some common encodings can bypass the codecs lookup machinery to improve performance. These optimization opportunities are only recognized by CPython for a limited set of (case insensitive) aliases : utf-8, utf8, latin-1, latin1, iso-8859-1, iso8859-1, mbcs (Windows only), ascii, us-ascii, utf-16, utf16, utf-32, utf32, and the same using underscores instead of dashes. Using alternative aliases for these encodings may result in slower execution.

Modifié dans la version 3.6 : Optimization opportunity recognized for us-ascii.

Many of the character sets support the same languages. They vary in individual characters (e.g. whether the EURO SIGN is supported or not), and in the assignment of characters to code positions. For the European languages in particular, the following variants typically exist :

— an ISO 8859 codeset

— a Microsoft Windows code page, which is typically derived from an 8859 codeset, but replaces control characters with additional graphic characters
— an IBM EBCDIC code page
— an IBM PC code page, which is ASCII compatible

Codec	Aliases	Languages
ascii	*646*, *us-ascii*	Anglais
big5	*big5-tw*, *csbig5*	Chinois Traditionnel
big5hkscs	*big5-hkscs*, *hkscs*	Chinois Traditionnel
cp037	*IBM037*, *IBM039*	Anglais
cp273	*273*, *IBM273*, *csIBM273*	Allemand Ajouté dans la version 3.4.
cp424	*EBCDIC-CP-HE*, *IBM424*	Hébreux
cp437	*437*, *IBM437*	Anglais
cp500	*EBCDIC-CP-BE*, *EBCDIC-CP-CH*, *IBM500*	Europe de l'ouest
cp720		Arabe
cp737		Grec
cp775	*IBM775*	Langues Baltiques
cp850	*850*, *IBM850*	Europe de l'ouest
cp852	*852*, *IBM852*	Europe centrale et Europe de l'Est
cp855	*855*, *IBM855*	Bulgare, Biélorusse, Macédonien, Russe, Serbe
cp856		Hébreux
cp857	*857*, *IBM857*	Turc
cp858	*858*, *IBM858*	Europe de l'ouest
cp860	*860*, *IBM860*	Portugais
cp861	*861*, *CP-IS*, *IBM861*	Islandais
cp862	*862*, *IBM862*	Hébreux
cp863	*863*, *IBM863*	Canadien
cp864	*IBM864*	Arabe
cp865	*865*, *IBM865*	Danish, Norwegian
cp866	*866*, *IBM866*	Russe
cp869	*869*, *CP-GR*, *IBM869*	Grec
cp874		Thai
cp875		Grec
cp932	932, ms932, mskanji, ms-kanji, windows-31j	Japanese
cp949	*949*, *ms949*, *uhc*	Korean
cp950	*950*, *ms950*	Chinois Traditionnel
cp1006		Urdu
cp1026	*ibm1026*	Turc
cp1125	*1125*, *ibm1125*, *cp866u*, *ruscii*	Ukrainian Ajouté dans la version 3.4.
cp1140	*ibm1140*	Europe de l'ouest
cp1250	*windows-1250*	Europe centrale et Europe de l'Est
cp1251	*windows-1251*	Bulgare, Biélorusse, Macédonien, Russe, Serbe
cp1252	*windows-1252*	Europe de l'ouest
cp1253	*windows-1253*	Grec
cp1254	*windows-1254*	Turc
cp1255	*windows-1255*	Hébreux
cp1256	*windows-1256*	Arabe
cp1257	*windows-1257*	Langues Baltiques
cp1258	*windows-1258*	Vietnamese
euc_jp	*eucjp*, *ujis*, *u-jis*	Japanese
euc_jis_2004	*jisx0213*, *eucjis2004*	Japanese

suite sur la page suivante

Tableau 1 – suite de la page précédente

Codec	Aliases	Languages
euc_jisx0213	*eucjisx0213*	Japanese
euc_kr	*euckr*, *korean*, *ksc5601*, *ks_c-5601*, *ks_c-5601-1987*, *ksx1001*, *ks_x-1001*	Korean
gb2312	*chinese*, *csiso58gb231280*, *euc-cn*, *euccn*, *eucgb2312-cn*, *gb2312-1980*, *gb2312-80*, *iso-ir-58*	Simplified Chinese
gbk	*936*, *cp936*, *ms936*	Unified Chinese
gb18030	*gb18030-2000*	Unified Chinese
hz	*hzgb*, *hz-gb*, *hz-gb-2312*	Simplified Chinese
iso2022_jp	*csiso2022jp*, *iso2022jp*, *iso-2022-jp*	Japanese
iso2022_jp_1	*iso2022jp-1*, *iso-2022-jp-1*	Japanese
iso2022_jp_2	*iso2022jp-2*, *iso-2022-jp-2*	Japanese, Korean, Simplified Chinese, Western Europe, Greek
iso2022_jp_2004	*iso2022jp-2004*, *iso-2022-jp-2004*	Japanese
iso2022_jp_3	*iso2022jp-3*, *iso-2022-jp-3*	Japanese
iso2022_jp_ext	*iso2022jp-ext*, *iso-2022-jp-ext*	Japanese
iso2022_kr	*csiso2022kr*, *iso2022kr*, *iso-2022-kr*	Korean
latin_1	*iso-8859-1*, *iso8859-1*, *8859*, *cp819*, *latin*, *latin1*, *L1*	Europe de l'ouest
iso8859_2	*iso-8859-2*, *latin2*, *L2*	Europe centrale et Europe de l'Est
iso8859_3	*iso-8859-3*, *latin3*, *L3*	Esperanto, Maltese
iso8859_4	*iso-8859-4*, *latin4*, *L4*	Langues Baltiques
iso8859_5	*iso-8859-5*, *cyrillic*	Bulgare, Biélorusse, Macédonien, Russe, Serbe
iso8859_6	*iso-8859-6*, *arabic*	Arabe
iso8859_7	*iso-8859-7*, *greek*, *greek8*	Grec
iso8859_8	*iso-8859-8*, *hebrew*	Hébreux
iso8859_9	*iso-8859-9*, *latin5*, *L5*	Turc
iso8859_10	*iso-8859-10*, *latin6*, *L6*	Nordic languages
iso8859_11	*iso-8859-11*, *thai*	Thai languages
iso8859_13	*iso-8859-13*, *latin7*, *L7*	Langues Baltiques
iso8859_14	*iso-8859-14*, *latin8*, *L8*	Celtic languages
iso8859_15	*iso-8859-15*, *latin9*, *L9*	Europe de l'ouest
iso8859_16	*iso-8859-16*, *latin10*, *L10*	South-Eastern Europe
johab	*cp1361*, *ms1361*	Korean
koi8_r		Russe
koi8_t		*Tajik* Ajouté dans la version 3.5.
koi8_u		Ukrainian
kz1048	*kz_1048*, *strk1048_2002*, *rk1048*	Kazakh Ajouté dans la version 3.5.
mac_cyrillic	*maccyrillic*	Bulgare, Biélorusse, Macédonien, Russe, Serbe
mac_greek	*macgreek*	Grec
mac_iceland	*maciceland*	Islandais
mac_latin2	maclatin2, maccentraleurope, mac_centeuro	Europe centrale et Europe de l'Est
mac_roman	*macroman*, *macintosh*	Europe de l'ouest
mac_turkish	*macturkish*	Turc
ptcp154	*csptcp154*, *pt154*, *cp154*, *cyrillic-asian*	Kazakh
shift_jis	*csshiftjis*, *shiftjis*, *sjis*, *s_jis*	Japanese

suite sur la page suivante

Tableau 1 – suite de la page précédente

Codec	Aliases	Languages
shift_jis_2004	*shiftjis2004*, *sjis_2004*, *sjis2004*	Japanese
shift_jisx0213	*shiftjisx0213*, *sjisx0213*, *s_jisx0213*	Japanese
utf_32	*U32*, *utf32*	all languages
utf_32_be	*UTF-32BE*	all languages
utf_32_le	*UTF-32LE*	all languages
utf_16	*U16*, *utf16*	all languages
utf_16_be	*UTF-16BE*	all languages
utf_16_le	*UTF-16LE*	all languages
utf_7	*U7*, *unicode-1-1-utf-7*	all languages
utf_8	U8, UTF, utf8, cp65001	all languages
utf_8_sig		all languages

Modifié dans la version 3.4 : The utf-16* and utf-32* encoders no longer allow surrogate code points (`U+D800`--`U+DFFF`) to be encoded. The utf-32* decoders no longer decode byte sequences that correspond to surrogate code points.

Modifié dans la version 3.8 : `cp65001` is now an alias to `utf_8`.

Modifié dans la version 3.14 : On Windows, `cpXXX` codecs are now available for all code pages.

7.2.4 Python Specific Encodings

A number of predefined codecs are specific to Python, so their codec names have no meaning outside Python. These are listed in the tables below based on the expected input and output types (note that while text encodings are the most common use case for codecs, the underlying codec infrastructure supports arbitrary data transforms rather than just text encodings). For asymmetric codecs, the stated meaning describes the encoding direction.

Text Encodings

The following codecs provide `str` to `bytes` encoding and *bytes-like object* to `str` decoding, similar to the Unicode text encodings.

Codec	Aliases	Signification
idna		Implement **RFC 3490**, see also `encodings.idna`. Only `errors='strict'` is supported.
mbcs	ansi, dbcs	Windows only : Encode the operand according to the ANSI codepage (CP_ACP).
oem		Windows only : Encode the operand according to the OEM codepage (CP_OEMCP). Ajouté dans la version 3.6.
palmos		Encoding of PalmOS 3.5.
punycode		Implement **RFC 3492**. Stateful codecs are not supported.
raw_unicode_escape		Latin-1 encoding with `\uXXXX` and `\UXXXXXXXX` for other code points. Existing backslashes are not escaped in any way. It is used in the Python pickle protocol.
undefined		Raise an exception for all conversions, even empty strings. The error handler is ignored.
unicode_escape		Encoding suitable as the contents of a Unicode literal in ASCII-encoded Python source code, except that quotes are not escaped. Decode from Latin-1 source code. Beware that Python source code actually uses UTF-8 by default.

Modifié dans la version 3.8 : ”unicode_internal” codec is removed.

Binary Transforms

The following codecs provide binary transforms : *bytes-like object* to `bytes` mappings. They are not supported by `bytes.decode()` (which only produces `str` output).

Codec	Aliases	Signification	Encoder / decoder
base64_codec [1]	base64, base_64	Convert the operand to multiline MIME base64 (the result always includes a trailing `'\n'`). Modifié dans la version 3.4 : accepts any *bytes-like object* as input for encoding and decoding	`base64.encodebytes()` / `base64.decodebytes()`
bz2_codec	bz2	Compress the operand using bz2.	`bz2.compress()` / `bz2.decompress()`
hex_codec	hex	Convert the operand to hexadecimal representation, with two digits per byte.	`binascii.b2a_hex()` / `binascii.a2b_hex()`
quopri_codec	quopri, quotedprintable, quoted_printable	Convert the operand to MIME quoted printable.	`quopri.encode()` with `quotetabs=True` / `quopri.decode()`
uu_codec	uu	Convert the operand using uuencode.	
zlib_codec	zip, zlib	Compress the operand using gzip.	`zlib.compress()` / `zlib.decompress()`

Ajouté dans la version 3.2 : Restoration of the binary transforms.

Modifié dans la version 3.4 : Restoration of the aliases for the binary transforms.

Text Transforms

The following codec provides a text transform : a `str` to `str` mapping. It is not supported by `str.encode()` (which only produces `bytes` output).

Codec	Aliases	Signification
rot_13	rot13	Return the Caesar-cypher encryption of the operand.

Ajouté dans la version 3.2 : Restoration of the `rot_13` text transform.

Modifié dans la version 3.4 : Restoration of the `rot13` alias.

7.2.5 `encodings.idna` --- Internationalized Domain Names in Applications

This module implements **RFC 3490** (Internationalized Domain Names in Applications) and **RFC 3492** (Nameprep : A Stringprep Profile for Internationalized Domain Names (IDN)). It builds upon the `punycode` encoding and `stringprep`.

If you need the IDNA 2008 standard from **RFC 5891** and **RFC 5895**, use the third-party idna module.

These RFCs together define a protocol to support non-ASCII characters in domain names. A domain name containing non-ASCII characters (such as `www.Alliancefrançaise.nu`) is converted into an ASCII-compatible encoding (ACE, such as `www.xn--alliancefranaise-npb.nu`). The ACE form of the domain name is then used in all places where arbitrary characters are not allowed by the protocol, such as DNS queries, HTTP `Host` fields, and so on. This conversion is carried out in the application ; if possible invisible to the user : The application should transparently convert Unicode domain labels to IDNA on the wire, and convert back ACE labels to Unicode before presenting them to the user.

1. In addition to *bytes-like objects*, `'base64_codec'` also accepts ASCII-only instances of `str` for decoding

Python supports this conversion in several ways : the `idna` codec performs conversion between Unicode and ACE, separating an input string into labels based on the separator characters defined in **section 3.1 of RFC 3490** and converting each label to ACE as required, and conversely separating an input byte string into labels based on the `.` separator and converting any ACE labels found into unicode. Furthermore, the *socket* module transparently converts Unicode host names to ACE, so that applications need not be concerned about converting host names themselves when they pass them to the socket module. On top of that, modules that have host names as function parameters, such as *http.client* and *ftplib*, accept Unicode host names (*http.client* then also transparently sends an IDNA hostname in the *Host* field if it sends that field at all).

When receiving host names from the wire (such as in reverse name lookup), no automatic conversion to Unicode is performed : applications wishing to present such host names to the user should decode them to Unicode.

The module *encodings.idna* also implements the nameprep procedure, which performs certain normalizations on host names, to achieve case-insensitivity of international domain names, and to unify similar characters. The nameprep functions can be used directly if desired.

`encodings.idna.`**`nameprep`**(*label*)

Return the nameprepped version of *label*. The implementation currently assumes query strings, so `AllowUnassigned` is true.

`encodings.idna.`**`ToASCII`**(*label*)

Convert a label to ASCII, as specified in **RFC 3490**. `UseSTD3ASCIIRules` is assumed to be false.

`encodings.idna.`**`ToUnicode`**(*label*)

Convert a label to Unicode, as specified in **RFC 3490**.

7.2.6 `encodings.mbcs` --- Windows ANSI codepage

This module implements the ANSI codepage (CP_ACP).

Availability : Windows.

Modifié dans la version 3.2 : Before 3.2, the *errors* argument was ignored ; `'replace'` was always used to encode, and `'ignore'` to decode.

Modifié dans la version 3.3 : Support any error handler.

7.2.7 `encodings.utf_8_sig` --- UTF-8 codec with BOM signature

This module implements a variant of the UTF-8 codec. On encoding, a UTF-8 encoded BOM will be prepended to the UTF-8 encoded bytes. For the stateful encoder this is only done once (on the first write to the byte stream). On decoding, an optional UTF-8 encoded BOM at the start of the data will be skipped.

CHAPITRE 8

Types de données

Les modules documentés dans ce chapitre fournissent une gamme de types de données spécialisés tel que les dates et les heures, les listes à type prédéfini, les *heap queue*, les queues synchronisées et les énumérations.

Python fournit aussi quelques types natifs, typiquement `dict`, `list`, `set`, `frozenset`, et `tuple`. La classe `str` est utilisée pour stocker des chaînes Unicode, et les classes `bytes` et `bytearray` des données binaires.

Les modules suivants sont documentés dans ce chapitre :

8.1 `datetime` — Types de base pour la date et l'heure

Code source : Lib/datetime.py

Le module `datetime` fournit des classes permettant de manipuler les dates et les heures.

Bien que les calculs de date et d'heure sont gérés, l'attention lors de l'implémentation s'est essentiellement portée sur l'efficacité de l'extraction des attributs en vue de leur manipulation et formatage pour l'affichage.

Astuce

Skip to *the format codes*.

Voir aussi

Module `calendar`
Fonctions génériques associées au calendrier.

Module `time`
Accès aux données d'horaires et aux conversions associées.

Module `zoneinfo`
Fuseaux horaires concrets représentant la base de données IANA.

Paquet dateutil
Bibliothèque tierce avec prise en charge complète du fuseau horaire et de l'analyse de dates sous forme textuelle.

Package DateType
Bibliothèque tierce qui introduit des types statiques distincts pour, par ex. permettre aux *vérificateurs statiques de typage* de faire la différence entre les dates et horaires naïfs et avisés.

8.1.1 Objets avisés et naïfs

Les objets *date* et *time* peuvent être classés comme « avisés » ou « naïfs » selon qu'ils contiennent ou non des informations sur le fuseau horaire.

Un objet **avisé** possède suffisamment de connaissance des règles à appliquer et des politiques d'ajustement de l'heure comme les informations sur les fuseaux horaires et l'heure d'été pour se situer de façon relative par rapport à d'autres objets avisés. Un objet avisé est utilisé pour représenter un instant précis qui n'est pas ouvert à l'interprétation [1].

Un objet **naïf** ne comporte pas assez d'informations pour se situer sans ambiguïté par rapport à d'autres objets *date/time*. Le fait qu'un objet naïf représente un Temps universel coordonné (UTC), une heure locale ou une heure dans un autre fuseau horaire dépend complètement du programme, tout comme un nombre peut représenter une longueur, un poids ou une distance pour le programme. Les objets naïfs sont simples à comprendre et il est aisé de travailler avec, au prix de négliger certains aspects de la réalité.

Pour les applications nécessitant des objets avisés, les objets `datetime` et `time` ont un attribut facultatif renseignant le fuseau horaire, `tzinfo`, qui peut être une instance d'une sous-classe de la classe abstraite `tzinfo`. Ces objets `tzinfo` regroupent des informations sur le décalage par rapport à l'heure UTC, le nom du fuseau horaire, et si l'heure d'été est en vigueur.

Notez qu'une seule classe concrète `tzinfo`, la classe `timezone`, est proposée par le module `datetime`. La classe `timezone` peut représenter des fuseaux horaires simples avec un décalage fixe par rapport à UTC, comme UTC lui-même ou les fuseaux EST et EDT d'Amérique du Nord. Gérer des fuseaux horaires d'un niveau de détails plus avancé est à la charge de l'application. Les règles d'ajustement du temps à travers le monde sont plus politiques que rationnelles, changent fréquemment, et il n'y a pas de standard qui vaille pour toute application, en dehors d'UTC.

8.1.2 Constantes

Le module `datetime` exporte les constantes suivantes :

`datetime.`**`MINYEAR`**
Le numéro d'année le plus petit autorisé dans un objet `date` ou `datetime`. `MINYEAR` vaut 1.

`datetime.`**`MAXYEAR`**
Le numéro d'année le plus grand autorisé dans un objet `date` ou `datetime`. `MAXYEAR` vaut 9999.

`datetime.`**`UTC`**
Alias du singleton de fuseau horaire UTC `datetime.timezone.utc`.

Ajouté dans la version 3.11.

8.1.3 Types disponibles

class `datetime.`**`date`**
Une date naïve idéalisée, en supposant que le calendrier Grégorien actuel a toujours existé et qu'il existera toujours. Attributs : `year`, `month` et `day`.

class `datetime.`**`time`**
Un temps idéalisé, indépendant d'une date particulière, en supposant qu'une journée est composée d'exactement 24*60*60 secondes (il n'y a pas ici de notion de « seconde intercalaire »). Attributs : `hour`, `minute`, `second`, `microsecond` et `tzinfo`.

class `datetime.`**`datetime`**
Une combinaison d'une date et d'une heure. Attributs : `year`, `month`, `day`, `hour`, `minute`, `second`, `microsecond`, et `tzinfo`.

1. Si on ignore les effets de la Relativité

class datetime.**timedelta**

Une durée qui exprime la différence entre deux instances de *datetime* ou *date* en microsecondes.

class datetime.**tzinfo**

Une classe mère abstraite pour les objets portants des informations sur les fuseaux horaires. Ceux-ci sont utilisés par les classes *datetime* et *time* pour donner une notion personnalisable d'ajustement d'horaire (par exemple la prise en compte d'un fuseau horaire et/ou de l'heure d'été).

class datetime.**timezone**

Une classe qui implémente la classe mère abstraite *tzinfo* en tant que décalage fixe par rapport au temps UTC.

Ajouté dans la version 3.2.

Les objets issus de ces types sont immuables.

Relations entre les sous-classes :

```
object
    timedelta
    tzinfo
        timezone
    time
    date
        datetime
```

Propriétés communes

Les types *date*, *datetime*, *time*, et *timezone* partagent les caractéristiques suivantes :

- Les objets issus de ces types sont immuables.
- Les objets de ces types sont *hachable*, ce qui signifie qu'ils peuvent être utilisés comme clés de dictionnaire.
- Les objets de ces types peuvent être sérialisés efficacement par le module *pickle*.

Catégorisation d'un objet en « avisé » ou « naïf »

Les objets de type *date* sont toujours naïfs.

Un objet du type *time* ou *datetime* peut être avisé ou naïf.

Un objet *datetime* `d` est avisé si les deux conditions suivantes vérifient :

1. `d.tzinfo` ne vaut pas `None`
2. `d.tzinfo.utcoffset(d)` ne renvoie pas `None`

Autrement, `d` est naïf.

Un objet *time* `t` est avisé si les deux conditions suivantes vérifient :

1. `t.tzinfo` ne vaut pas `None`
2. `t.tzinfo.utcoffset(None)` ne renvoie pas `None`.

Autrement, `t` est naïf.

La distinction entre avisé et naïf ne s'applique pas aux objets de type *timedelta*.

8.1.4 Objets timedelta

Un objet *timedelta* représente une durée, c'est-à-dire la différence entre deux instances de *datetime* ou *date*.

class datetime.**timedelta**(*days=0*, *seconds=0*, *microseconds=0*, *milliseconds=0*, *minutes=0*, *hours=0*, *weeks=0*)

Tous les arguments sont optionnels et ont 0 comme valeur par défaut. Les arguments peuvent être des entiers ou des flottants et ils peuvent être positifs ou négatifs.

Seuls les *jours*, les *secondes* et les *microsecondes* sont stockés en interne. Les arguments sont convertis dans ces unités :

— Une milliseconde est convertie en 1000 microsecondes.
— Une minute est convertie en 60 secondes.
— Une heure est convertie en 3600 secondes.
— Une semaine est convertie en 7 jours.

et ensuite les jours, secondes et microsecondes sont normalisés pour que la représentation soit unique avec

— `0 <= microseconds < 1000000`
— `0 <= secondes < 3600*24` (le nombre de secondes dans une journée)
— `-999999999 <= days <= 999999999`

L'exemple suivant illustre comment tous les arguments autres que *days*, *seconds* et *microseconds* sont « fusionnés » et normalisés dans ces trois attributs résultants :

```
>>> from datetime import timedelta
>>> delta = timedelta(
...     days=50,
...     seconds=27,
...     microseconds=10,
...     milliseconds=29000,
...     minutes=5,
...     hours=8,
...     weeks=2
... )
>>> # Only days, seconds, and microseconds remain
>>> delta
datetime.timedelta(days=64, seconds=29156, microseconds=10)
```

Si l'un des arguments est un flottant et qu'il y a des microsecondes décimales, les microsecondes décimales laissées par les arguments sont combinées et leur somme est arrondie à la microseconde la plus proche en arrondissant les demis vers le nombre pair. Si aucun argument n'est flottant, les processus de conversion et de normalisation seront exacts (pas d'informations perdues).

Si la valeur normalisée des jours déborde de l'intervalle indiqué, une `OverflowError` est levée.

Notez que la normalisation de valeurs négatives peut être surprenante au premier abord. Par exemple :

```
>>> from datetime import timedelta
>>> d = timedelta(microseconds=-1)
>>> (d.days, d.seconds, d.microseconds)
(-1, 86399, 999999)
```

Attributs de la classe :

`timedelta.`**`min`**

L'objet `timedelta` le plus négatif, `timedelta(-999999999)`.

`timedelta.`**`max`**

L'objet `timedelta` le plus positif, `timedelta(days=999999999, hours=23, minutes=59, seconds=59, microseconds=999999)`.

`timedelta.`**`resolution`**

La plus petite différence entre des objets `timedelta` non égaux, `timedelta(microseconds=1)`.

Il est à noter, du fait de la normalisation, que `timedelta.max` est plus grand que `-timedelta.min`. `-timedelta.max` n'est pas représentable sous la forme d'un objet `timedelta`.

Attributs de l'instance (en lecture seule) :

`timedelta.`**`days`**

Entre -999999999 et 999999999 inclus

`timedelta.`**`seconds`**

Entre 0 et 86399 inclus.

Prudence

It is a somewhat common bug for code to unintentionally use this attribute when it is actually intended to get a *total_seconds()* value instead :

```
>>> from datetime import timedelta
>>> duration = timedelta(seconds=11235813)
>>> duration.days, duration.seconds
(130, 3813)
>>> duration.total_seconds()
11235813.0
```

`timedelta.`**`microseconds`**

Entre 0 et 999999 inclus.

Opérations gérées :

Opération	Résultat
`t1 = t2 + t3`	Somme de `t2` et `t3`. Ensuite `t1 - t2 == t3` et `t1 - t3 == t2` sont des expressions vraies. (1)
`t1 = t2 - t3`	Différence entre `t2` et `t3`. Ensuite `t1 == t2 - t3` et `t2 == t1 + t3` sont des expressions vraies. (1)(6)
`t1 = t2 * i` or `t1 = i * t2`	Delta multiplié par un entier. Ensuite `t1 // i == t2` est vrai, en admettant que `i != 0`.
	De manière générale, `t1 * i == t1 * (i-1) + t1` est vrai. (1)
`t1 = t2 * f` or `t1 = f * t2`	Delta multiplié par un flottant. Le résultat est arrondi au multiple le plus proche de `timedelta.resolution` en utilisant la règle de l'arrondi au pair le plus proche.
`f = t2 / t3`	Division (3) de la durée totale `t2` par l'unité d'intervalle `t3`. Renvoie un objet *float*.
`t1 = t2 / f` or `t1 = t2 / i`	Delta divisé par un flottant ou un entier. Le résultat est arrondi au multiple le plus proche de `timedelta.resolution` en utilisant la règle de l'arrondi au pair le plus proche.
`t1 = t2 // i` or `t1 = t2 // t3`	Le quotient est calculé et le reste (s'il y en a un) est ignoré. Dans le second cas, un entier est renvoyé. (3)
`t1 = t2 % t3`	Le reste est calculé comme un objet de type *timedelta*. (3)
`q, r = divmod(t1, t2)`	Calcule le quotient et le reste : `q = t1 // t2` (3) et `r = t1 % t2`. `q` est un entier et `r` est un objet *timedelta*.
`+t1`	Renvoie un objet *timedelta* avec la même valeur. (2)
`-t1`	Équivalent à `timedelta(-t1.days, -t1.seconds, -t1.microseconds)`, et à `t1 * -1`. (1)(4)
`abs(t)`	Équivalent à `+t` quand `t.days >= 0`, et à `-t` quand `t.days < 0`. (2)
`str(t)`	Renvoie une chaîne de la forme `[D day[s], ][H]H:MM:SS[.UUUUUU]`, où D est négatif pour `t` négatif. (5)
`repr(t)`	Renvoie une chaîne de la forme objet *timedelta* comme un appel construit avec des valeurs d'attributs canoniques.

Notes :

(1) Ceci est exact, mais peut provoquer un débordement.

(2) Ceci est exact, et ne peut pas provoquer un débordement.

(3) Une division par zéro lève une *ZeroDivisionError*.

(4) `-timedelta.max` n'est pas représentable avec un objet *timedelta*.

(5) La représentation en chaîne des objets *timedelta* est normalisée similairement à leur représentation interne. Cela amène à des résultats inhabituels pour des *timedeltas* négatifs. Par exemple :

```
>>> timedelta(hours=-5)
datetime.timedelta(days=-1, seconds=68400)
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> print(_)
-1 day, 19:00:00
```

(6) L'expression `t2 - t3` est toujours égale à l'expression `t2 + (-t3)` sauf si `t3` vaut `timedelta.max` ; dans ce cas, la première expression produit une valeur alors que la seconde produit un débordement.

En plus des opérations listées ci-dessus, les objets *timedelta* implémentent certaines additions et soustractions avec des objets *date* et *datetime* (voir ci-dessous).

Modifié dans la version 3.2 : La division entière et la vraie division d'un objet *timedelta* par un autre *timedelta* sont maintenant gérées, comme le sont les opérations de reste euclidien et la fonction *divmod()*. La vraie division et la multiplication d'un objet *timedelta* par un *float* sont maintenant implémentées.

Les objets *timedelta* supportent les comparaisons d'égalité et d'ordre.

Dans les contextes booléens, un objet *timedelta* est considéré comme vrai si et seulement s'il n'est pas égal à `timedelta(0)`.

Méthodes de l'instance :

`timedelta.`**`total_seconds`**`()`

Renvoie le nombre total de secondes contenues dans la durée. Équivalent à `td / timedelta(seconds=1)`. Pour un intervalle dont l'unité n'est pas la seconde, utilisez directement la division (par exemple, `td / timedelta(microseconds=1)`).

Notez que pour des intervalles de temps très larges (supérieurs à 270 ans sur la plupart des plateformes), cette méthode perdra la précision des microsecondes.

Ajouté dans la version 3.2.

Exemples d'utilisation de la classe `timedelta` :

Un exemple supplémentaire de normalisation :

```
>>> # Components of another_year add up to exactly 365 days
>>> from datetime import timedelta
>>> year = timedelta(days=365)
>>> another_year = timedelta(weeks=40, days=84, hours=23,
...                          minutes=50, seconds=600)
>>> year == another_year
True
>>> year.total_seconds()
31536000.0
```

Exemples d'arithmétique avec la classe *timedelta* :

```
>>> from datetime import timedelta
>>> year = timedelta(days=365)
>>> ten_years = 10 * year
>>> ten_years
datetime.timedelta(days=3650)
>>> ten_years.days // 365
10
>>> nine_years = ten_years - year
>>> nine_years
datetime.timedelta(days=3285)
>>> three_years = nine_years // 3
>>> three_years, three_years.days // 365
(datetime.timedelta(days=1095), 3)
```

8.1.5 Objets `date`

Un objet `date` représente une date (année, mois et jour) dans un calendrier idéal, le calendrier grégorien actuel étant indéfiniment étendu dans les deux sens.

Le 1 er janvier de l'année 1 est appelé jour numéro 1, le 2 janvier de l'année 1 est appelé jour numéro 2, et ainsi de suite. [2]

class `datetime.`**`date`**(*year*, *month*, *day*)

Tous les arguments sont requis. Les arguments peuvent être des entiers, dans les intervalles suivants :

— `MINYEAR <= year <= MAXYEAR`

— `1 <= month <= 12`

— `1 <= day <= nombre de jours dans le mois et l'année donnés`

Si un argument est donné en dehors de ces intervalles, une `ValueError` est levée.

Autres constructeurs, méthodes de classe :

classmethod `date.`**`today`**()

Renvoie la date locale courante.

Cela est équivalent à `date.fromtimestamp(time.time())`.

classmethod `date.`**`fromtimestamp`**(*timestamp*)

Renvoie la date locale correspondant à l'horodatage POSIX, telle que renvoyée par `time.time()`.

Elle peut lever une `OverflowError`, si l'horodatage est en dehors des bornes gérées par la fonction C `localtime()` de la plateforme, et une exception `OSError` en cas d'échec de `localtime()`. Il est commun d'être restreint aux années entre 1970 et 2038. Notez que sur les systèmes non *POSIX* qui incluent les secondes intercalaires dans leur notion d'horodatage, ces secondes sont ignorées par `fromtimestamp()`.

Modifié dans la version 3.3 : Lève une `OverflowError` plutôt qu'une `ValueError` si l'horodatage (*timestamp* en anglais) est en dehors des bornes gérées par la fonction C `localtime()` de la plateforme. Lève une `OSError` plutôt qu'une `ValueError` en cas d'échec de `localtime()`.

classmethod `date.`**`fromordinal`**(*ordinal*)

Renvoie la date correspondant à l'ordinal grégorien proleptique, où le 1er janvier de l'an 1 a l'ordinal 1.

`ValueError` est levée à moins que `1 <= ordinal <= date.max.toordinal()`. Pour toute date `d`, `date.fromordinal(d.toordinal()) == d`.

classmethod `date.`**`fromisoformat`**(*date_string*)

Renvoie une `date` correspondant à *date_string* dans un format ISO 8601 valide, avec les exceptions suivantes :

1. Les dates à précision réduite ne sont actuellement pas gérées (`YYYY-MM`, `YYYY`).
2. Extended date representations are not currently supported (`±YYYYYY-MM-DD`).
3. Ordinal dates are not currently supported (`YYYY-OOO`).

Exemples :

```
>>> from datetime import date
>>> date.fromisoformat('2019-12-04')
datetime.date(2019, 12, 4)
>>> date.fromisoformat('20191204')
datetime.date(2019, 12, 4)
>>> date.fromisoformat('2021-W01-1')
datetime.date(2021, 1, 4)
```

Ajouté dans la version 3.7.

Modifié dans la version 3.11 : Auparavant, cette méthode prenait en charge seulement le format `YYYY-MM-DD`.

2. Cela correspond à la définition du calendrier « grégorien proleptique » dans le livre *Calendrical Calculations* de Dershowitz et Reingold, où il est la base de tous les calculs. Référez-vous au livre pour les algorithmes de conversion entre calendriers grégorien proleptique et les autres systèmes.

classmethod date.**fromisocalendar**(*year*, *week*, *day*)

Renvoie une `date` correspondant à la date du calendrier ISO définie par l'année, la semaine et le jour. C'est la réciproque de la fonction `date.isocalendar()`.

Ajouté dans la version 3.8.

classmethod date.**strptime**(*date_string*, *format*)

Return a `date` corresponding to *date_string*, parsed according to *format*. This is equivalent to :

```
date(*(time.strptime(date_string, format)[0:3]))
```

`ValueError` is raised if the date_string and format can't be parsed by `time.strptime()` or if it returns a value which isn't a time tuple. See also *strftime() and strptime() Behavior* and `date.fromisoformat()`.

Note

If *format* specifies a day of month without a year a `DeprecationWarning` is emitted. This is to avoid a quadrennial leap year bug in code seeking to parse only a month and day as the default year used in absence of one in the format is not a leap year. Such *format* values may raise an error as of Python 3.15. The workaround is to always include a year in your *format*. If parsing *date_string* values that do not have a year, explicitly add a year that is a leap year before parsing :

```
>>> from datetime import date
>>> date_string = "02/29"
>>> when = date.strptime(f"{date_string};1984", "%m/%d;%Y")  # Avoids leap␣
↪year bug.
>>> when.strftime("%B %d")
'February 29'
```

Ajouté dans la version 3.14.

Attributs de la classe :

date.**min**

La plus vieille date représentable, `date(MINYEAR, 1, 1)`.

date.**max**

La dernière date représentable, `date(MAXYEAR, 12, 31)`.

date.**resolution**

La plus petite différence possible entre deux objets dates non-égaux, `timedelta(days=1)`.

Attributs de l'instance (en lecture seule) :

date.**year**

Entre `MINYEAR` et `MAXYEAR` inclus.

date.**month**

Entre 1 et 12 inclus.

date.**day**

Entre 1 et le nombre de jours du mois donné de l'année donnée.

Opérations gérées :

Opération	Résultat
`date2 = date1 + timedelta`	`date2` will be `timedelta.days` days after `date1`. (1)
`date2 = date1 - timedelta`	Computes `date2` such that `date2 + timedelta == date1`. (2)
`timedelta = date1 - date2`	(3)
`date1 == date2` `date1 != date2`	Equality comparison. (4)

`date1 < date2`
`date1 > date2`
`date1 <= date2`
`date1 >= date2`
	Order comparison. (5)

Notes :

(1) `date2` est déplacée en avant dans le temps si `timedelta.days > 0`, ou en arrière si `timedelta.days < 0`. Après quoi `date2 - date1 == timedelta.days`. `timedelta.seconds` et `timedelta.microseconds` sont ignorés. Une `OverflowError` est levée si `date2.year` devait être inférieure à `MINYEAR` ou supérieure à `MAXYEAR`.

(2) `timedelta.seconds` et `timedelta.microseconds` sont ignorés.

(3) This is exact, and cannot overflow. `timedelta.seconds` and `timedelta.microseconds` are 0, and `date2 + timedelta == date1` after.

(4) `date` objects are equal if they represent the same date.

`date` objects that are not also `datetime` instances are never equal to `datetime` objects, even if they represent the same date.

(5) *date1* is considered less than *date2* when *date1* precedes *date2* in time. In other words, `date1 < date2` if and only if `date1.toordinal() < date2.toordinal()`.

Order comparison between a `date` object that is not also a `datetime` instance and a `datetime` object raises `TypeError`.

Modifié dans la version 3.13 : Comparison between `datetime` object and an instance of the `date` subclass that is not a `datetime` subclass no longer converts the latter to `date`, ignoring the time part and the time zone. The default behavior can be changed by overriding the special comparison methods in subclasses.

Dans des contextes booléens, tous les objets `date` sont considérés comme vrai.

Méthodes de l'instance :

`date.`**`replace`**(*year=self.year*, *month=self.month*, *day=self.day*)

Renvoie une date avec la même valeur, à l'exception des paramètres nommés pour lesquels une nouvelle valeur est donnée en argument.

Exemple :

```
>>> from datetime import date
>>> d = date(2002, 12, 31)
>>> d.replace(day=26)
datetime.date(2002, 12, 26)
```

`date` objects are also supported by generic function `copy.replace()`.

`date.`**`timetuple`**()

Renvoie une `time.struct_time` telle que renvoyée par `time.localtime()`.

Les heures, minutes et secondes sont égales à 0 et le drapeau DST vaut -1.

`d.timetuple()` est équivalent à :

```
time.struct_time((d.year, d.month, d.day, 0, 0, 0, d.weekday(), yday, -1))
```

where `yday = d.toordinal() - date(d.year, 1, 1).toordinal() + 1` is the day number within the current year starting with 1 for January 1st.

`date.toordinal()`

Return the proleptic Gregorian ordinal of the date, where January 1 of year 1 has ordinal 1. For any *date* object `d`, `date.fromordinal(d.toordinal()) == d`.

`date.weekday()`

Renvoie le jour de la semaine sous forme de nombre, où lundi vaut 0 et dimanche vaut 6. Par exemple, `date(2002, 12, 4).weekday() == 2`, un mercredi. Voir aussi *isoweekday()*.

`date.isoweekday()`

Renvoie le jour de la semaine sous forme de nombre, où lundi vaut 1 et dimanche vaut 7. Par exemple, `date(2002, 12, 4).isoweekday() == 3`, un mercredi. Voir aussi *weekday()*, *isocalendar()*.

`date.isocalendar()`

Renvoie un objet *named tuple* avec trois composants : `year`, `week` et `weekday`.

Le calendrier ISO est une variante largement utilisée du calendrier grégorien. [3]

Une année ISO est composée de 52 ou 53 semaines pleines, où chaque semaine débute un lundi et se termine un dimanche. La première semaine d'une année ISO est la première semaine calendaire (grégorienne) de l'année comportant un jeudi. Elle est appelée la semaine numéro 1, et l'année ISO de ce jeudi est la même que son année Grégorienne.

Par exemple, l'année 2004 débute un jeudi, donc la première semaine de l'année ISO 2004 débute le lundi 29 décembre 2003 et se termine le dimanche 4 janvier 2004 :

```
>>> from datetime import date
>>> date(2003, 12, 29).isocalendar()
datetime.IsoCalendarDate(year=2004, week=1, weekday=1)
>>> date(2004, 1, 4).isocalendar()
datetime.IsoCalendarDate(year=2004, week=1, weekday=7)
```

Modifié dans la version 3.9 : Le résultat a changé d'un *n*-uplet à un *named tuple*.

`date.isoformat()`

Renvoie une chaîne représentant la date au format ISO 8601, `YYYY-MM-DD` :

```
>>> from datetime import date
>>> date(2002, 12, 4).isoformat()
'2002-12-04'
```

`date.__str__()`

For a date `d`, `str(d)` is equivalent to `d.isoformat()`.

`date.ctime()`

Renvoie une chaîne représentant la date :

```
>>> from datetime import date
>>> date(2002, 12, 4).ctime()
'Wed Dec  4 00:00:00 2002'
```

`d.ctime()` est équivalent à :

3. Voir R. H. van Gent guide des mathématiques du calendrier ISO 8601 pour une bonne explication.

```
time.ctime(time.mktime(d.timetuple()))
```

sur les plateformes où la fonction C native `ctime()` (que `time.ctime()` invoque, mais pas `date.ctime()`) est conforme au standard C.

`date.`**`strftime`**(*format*)

Renvoie une chaîne représentant la date, contrôlée par une chaîne de formatage explicite. Les codes de formatage se référant aux heures, minutes ou secondes auront pour valeur 0. Voir *strftime() and strptime() Behavior* et `date.isoformat()`.

`date.`**`__format__`**(*format*)

Identique à `date.strftime()`. Cela permet de spécifier une chaîne de formatage pour un objet `date` dans une chaîne de formatage littérale et à l'utilisation de `str.format()`. Voir *strftime() and strptime() Behavior* et `date.isoformat()`.

Exemple d'utilisation de la classe `date` :

Exemple de décompte des jours avant un évènement :

```
>>> import time
>>> from datetime import date
>>> today = date.today()
>>> today
datetime.date(2007, 12, 5)
>>> today == date.fromtimestamp(time.time())
True
>>> my_birthday = date(today.year, 6, 24)
>>> if my_birthday < today:
...     my_birthday = my_birthday.replace(year=today.year + 1)
...
>>> my_birthday
datetime.date(2008, 6, 24)
>>> time_to_birthday = abs(my_birthday - today)
>>> time_to_birthday.days
202
```

Plus d'exemples avec la classe `date` :

```
>>> from datetime import date
>>> d = date.fromordinal(730920) # 730920th day after 1. 1. 0001
>>> d
datetime.date(2002, 3, 11)

>>> # Methods related to formatting string output
>>> d.isoformat()
'2002-03-11'
>>> d.strftime("%d/%m/%y")
'11/03/02'
>>> d.strftime("%A %d. %B %Y")
'Monday 11. March 2002'
>>> d.ctime()
'Mon Mar 11 00:00:00 2002'
>>> 'The {1} is {0:%d}, the {2} is {0:%B}.'.format(d, "day", "month")
'The day is 11, the month is March.'

>>> # Methods for to extracting 'components' under different calendars
>>> t = d.timetuple()
>>> for i in t:
```

(suite sur la page suivante)

(suite de la page précédente)

```
...     print(i)
2002                # year
3                   # month
11                  # day
0
0
0
0                   # weekday (0 = Monday)
70                  # 70th day in the year
-1
>>> ic = d.isocalendar()
>>> for i in ic:
...     print(i)
2002                # ISO year
11                  # ISO week number
1                   # ISO day number ( 1 = Monday )

>>> # A date object is immutable; all operations produce a new object
>>> d.replace(year=2005)
datetime.date(2005, 3, 11)
```

8.1.6 Objets `datetime`

Un objet `datetime` est un seul et même objet contenant toute l'information d'un objet `date` et d'un objet `time`.

Comme un objet `date`, un objet `datetime` utilise le calendrier Grégorien actuel étendu vers le passé et le futur ; comme un objet `time`, un objet `datetime` suppose qu'il y a exactement 3600*24 secondes chaque jour.

Constructeur :

class datetime.**datetime**(*year*, *month*, *day*, *hour=0*, *minute=0*, *second=0*, *microsecond=0*, *tzinfo=None*, *, *fold=0*)

Les arguments *year*, *month* et *day* sont requis. *tzinfo* peut être `None` ou une instance d'une sous-classe de `tzinfo`. Les arguments restant doivent être des nombres, dans les intervalles suivants :

- `MINYEAR <= year <= MAXYEAR`,
- `1 <= month <= 12`,
- `1 <= day <= nombre de jours dans le mois donné de l'année donnée`,
- `0 <= hour < 24`,
- `0 <= minute < 60`,
- `0 <= second < 60`,
- `0 <= microsecond < 1000000`,
- `fold in [0, 1]`.

Si un argument est donné en dehors de ces intervalles, une `ValueError` est levée.

Modifié dans la version 3.6 : Added the *fold* parameter.

Autres constructeurs, méthodes de classe :

classmethod datetime.**today**()

Return the current local date and time, with `tzinfo` `None`.

Équivalent à :

```
datetime.fromtimestamp(time.time())
```

Voir aussi `now()`, `fromtimestamp()`.

Cette méthode est fonctionnellement équivalente à `now()`, mais sans le paramètre `tz`.

classmethod datetime.**now**(*tz=None*)

Renvoie la date et l'heure locale actuelle.

Si l'argument optionnel *tz* est `None` ou n'est pas spécifié, la méthode est similaire à `today()`, mais, si possible, apporte plus de précisions que ce qui peut être trouvé à travers un horodatage `time.time()` (par exemple, cela peut être possible sur des plateformes fournissant la fonction C `gettimeofday()`).

Si *tz* n'est pas `None`, il doit être une instance d'une sous-classe `tzinfo`, et la date et l'heure courantes sont converties vers le fuseau horaire *tz*.

Cette fonction est préférée à `today()` et `utcnow()`.

> **Note**
>
> Subsequent calls to `datetime.now()` may return the same instant depending on the precision of the underlying clock.

classmethod datetime.**utcnow**()

Renvoie la date et l'heure UTC actuelle, avec `tzinfo` `None`.

C'est semblable à `now()`, mais renvoie la date et l'heure UTC courantes, comme un objet `datetime` naïf. Un `datetime` UTC courant avisé peut être obtenu en appelant `datetime.now(timezone.utc)`. Voir aussi `now()`.

> **Avertissement**
>
> Parce que les objets naïfs `datetime` sont traités par de nombreuses méthodes `datetime` comme des heures locales, il est préférable d'utiliser les dates connues pour représenter les heures en UTC. En tant que tel, le moyen recommandé pour créer un objet représentant l'heure actuelle en UTC est d'appeler `datetime.now(timezone.utc)`.

Obsolète depuis la version 3.12 : Use `datetime.now()` with `UTC` instead.

classmethod datetime.**fromtimestamp**(*timestamp*, *tz=None*)

Renvoie la date et l'heure locales correspondant à l'horodatage (*timestamp* en anglais) *POSIX*, comme renvoyé par `time.time()`. Si l'argument optionnel *tz* est `None` ou n'est pas spécifié, l'horodatage est converti vers la date et l'heure locales de la plateforme, et l'objet `datetime` renvoyé est naïf.

Si *tz* n'est pas `None`, il doit être une instance d'une sous-classe `tzinfo`, et l'horodatage (*timestamp* en anglais) est converti vers le fuseau horaire *tz*.

`fromtimestamp()` peut lever une `OverflowError`, si l'horodatage est en dehors de l'intervalle de valeurs gérées par les fonctions C `localtime()` ou `gmtime()` de la plateforme, et une `OSError` en cas d'échec de `localtime()` ou `gmtime()`. Il est courant d'être restreint aux années de 1970 à 2038. Notez que sur les systèmes non *POSIX* qui incluent les secondes intercalaires dans leur notion d'horodatage, les secondes intercalaires sont ignorées par `fromtimestamp()`, et il est alors possible d'avoir deux horodatages différant d'une seconde produisant un objet `datetime` identique. Cette méthode est préférée à `utcfromtimestamp()`.

Modifié dans la version 3.3 : Lève une `OverflowError` plutôt qu'une `ValueError` si l'horodatage est en dehors de l'intervalle de valeurs gérées par les fonctions C `localtime()` ou `gmtime()` de la plateforme. Lève une `OSError` plutôt qu'une `ValueError` en cas d'échec de `localtime()` ou `gmtime()`.

Modifié dans la version 3.6 : `fromtimestamp()` peut renvoyer des instances avec l'attribut `fold` à 1.

classmethod datetime.**utcfromtimestamp**(*timestamp*)

Renvoie la classe UTC `datetime` correspondant à l'horodatage POSIX, avec `tzinfo` `None` (l'objet résultant est naïf).

Cela peut lever une `OverflowError`, si l'horodatage est en dehors de l'intervalle de valeurs gérées par la fonction C `gmtime()` de la plateforme, et une `OSError` en cas d'échec de `gmtime()`. Il est courant d'être restreint aux années de 1970 à 2038.

Pour obtenir un objet *datetime* avisé, appelez *fromtimestamp()* :

```
datetime.fromtimestamp(timestamp, timezone.utc)
```

Sur les plateformes respectant *POSIX*, cela est équivalent à l'expression suivante :

```
datetime(1970, 1, 1, tzinfo=timezone.utc) + timedelta(seconds=timestamp)
```

excepté que la dernière formule gère l'intervalle complet des années entre *MINYEAR* et *MAXYEAR* incluses.

> **Avertissement**
>
> Parce que les objets naïfs `datetime` sont traités par de nombreuses méthodes `datetime` comme des heures locales, il est préférable d'utiliser les dates connues pour représenter les heures en UTC. En tant que tel, le moyen recommandé pour créer un objet représentant un horodatage spécifique en UTC est d'appeler `datetime.fromtimestamp(timestamp, tz=timezone.utc)`.

Modifié dans la version 3.3 : Lève une *OverflowError* plutôt qu'une *ValueError* si l'horodatage est en dehors de l'intervalle de valeurs gérées par la fonction C `gmtime()` de la plateforme. Lève une *OSError* plutôt qu'une *ValueError* en cas d'échec de `gmtime()`.

Obsolète depuis la version 3.12 : Use *datetime.fromtimestamp()* with *UTC* instead.

classmethod datetime.**fromordinal**(*ordinal*)

Renvoie le *datetime* correspondant à l'ordinal du calendrier grégorien proleptique, où le 1er janvier de l'an 1 a l'ordinal 1. Une *ValueError* est levée à moins que `1 <= ordinal <= datetime.max.toordinal()`. Les heures, minutes, secondes et microsecondes du résultat valent toutes 0, et *tzinfo* est `None`.

classmethod datetime.**combine**(*date*, *time*, *tzinfo=time.tzinfo*)

Return a new *datetime* object whose date components are equal to the given *date* object's, and whose time components are equal to the given *time* object's. If the *tzinfo* argument is provided, its value is used to set the *tzinfo* attribute of the result, otherwise the *tzinfo* attribute of the *time* argument is used. If the *date* argument is a *datetime* object, its time components and *tzinfo* attributes are ignored.

For any *datetime* object `d`, `d == datetime.combine(d.date(), d.time(), d.tzinfo)`.

Modifié dans la version 3.6 : Ajout de l'argument *tzinfo*.

classmethod datetime.**fromisoformat**(*date_string*)

Renvoie une classe *datetime* correspondant à *date_string* dans un format ISO 8601 valide, avec les exceptions suivantes :

1. Les décalages de fuseaux horaires peuvent comporter des fractions de secondes.
2. Le séparateur `T` peut être remplacé par n'importe quel caractère Unicode.
3. Les fractions d'heures et de minutes ne sont pas gérées.
4. Les dates à précision réduite ne sont actuellement pas gérées (`YYYY-MM`, `YYYY`).
5. Extended date representations are not currently supported (`±YYYYYY-MM-DD`).
6. Ordinal dates are not currently supported (`YYYY-OOO`).

Exemples :

```
>>> from datetime import datetime
>>> datetime.fromisoformat('2011-11-04')
datetime.datetime(2011, 11, 4, 0, 0)
>>> datetime.fromisoformat('20111104')
datetime.datetime(2011, 11, 4, 0, 0)
>>> datetime.fromisoformat('2011-11-04T00:05:23')
datetime.datetime(2011, 11, 4, 0, 5, 23)
>>> datetime.fromisoformat('2011-11-04T00:05:23Z')
datetime.datetime(2011, 11, 4, 0, 5, 23, tzinfo=datetime.timezone.utc)
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> datetime.fromisoformat('20111104T000523')
datetime.datetime(2011, 11, 4, 0, 5, 23)
>>> datetime.fromisoformat('2011-W01-2T00:05:23.283')
datetime.datetime(2011, 1, 4, 0, 5, 23, 283000)
>>> datetime.fromisoformat('2011-11-04 00:05:23.283')
datetime.datetime(2011, 11, 4, 0, 5, 23, 283000)
>>> datetime.fromisoformat('2011-11-04 00:05:23.283+00:00')
datetime.datetime(2011, 11, 4, 0, 5, 23, 283000, tzinfo=datetime.timezone.utc)
>>> datetime.fromisoformat('2011-11-04T00:05:23+04:00')
datetime.datetime(2011, 11, 4, 0, 5, 23,
    tzinfo=datetime.timezone(datetime.timedelta(seconds=14400)))
```

Ajouté dans la version 3.7.

Modifié dans la version 3.11 : Previously, this method only supported formats that could be emitted by `date.isoformat()` or `datetime.isoformat()`.

classmethod datetime.**fromisocalendar**(*year*, *week*, *day*)

Renvoie une classe `datetime` correspondant à la date du calendrier ISO spécifiée par année, semaine et jour. Les composantes ne relevant pas de la date de *datetime* sont renseignées avec leurs valeurs par défaut normales. C'est la réciproque de la fonction `datetime.isocalendar()`.

Ajouté dans la version 3.8.

classmethod datetime.**strptime**(*date_string*, *format*)

Renvoie une classe `datetime` correspondant à *date_string*, analysée selon *format*.

If *format* does not contain microseconds or time zone information, this is equivalent to :

```
datetime(*(time.strptime(date_string, format)[0:6]))
```

Une `ValueError` est levée si *date_string* et *format* ne peuvent être analysés par `time.strptime()` ou si elle renvoie une valeur qui n'est pas un *n*-uplet de temps. Voir *strftime() and strptime() Behavior* et `datetime.fromisoformat()`.

Modifié dans la version 3.13 : If *format* specifies a day of month without a year a `DeprecationWarning` is now emitted. This is to avoid a quadrennial leap year bug in code seeking to parse only a month and day as the default year used in absence of one in the format is not a leap year. Such *format* values may raise an error as of Python 3.15. The workaround is to always include a year in your *format*. If parsing *date_string* values that do not have a year, explicitly add a year that is a leap year before parsing :

```
>>> from datetime import datetime
>>> date_string = "02/29"
>>> when = datetime.strptime(f"{date_string};1984", "%m/%d;%Y")  # Avoids leap␣
↪year bug.
>>> when.strftime("%B %d")
'February 29'
```

Attributs de la classe :

datetime.**min**

Le plus ancien `datetime` représentable, `datetime(MINYEAR, 1, 1, tzinfo=None)`.

datetime.**max**

Le dernier `datetime` représentable, `datetime(MAXYEAR, 12, 31, 23, 59, 59, 999999, tzinfo=None)`.

datetime.**resolution**

La plus petite différence possible entre deux objets `datetime` non-égaux, `timedelta(microseconds=1)`.

Attributs de l'instance (en lecture seule) :

`datetime.year`

Entre `MINYEAR` et `MAXYEAR` inclus.

`datetime.month`

Entre 1 et 12 inclus.

`datetime.day`

Entre 1 et le nombre de jours du mois donné de l'année donnée.

`datetime.hour`

Dans `range(24)`.

`datetime.minute`

Dans `range(60)`.

`datetime.second`

Dans `range(60)`.

`datetime.microsecond`

Dans `range(1000000)`.

`datetime.tzinfo`

L'objet passé en tant que paramètre *tzinfo* du constructeur de la classe `datetime` ou `None` si aucun n'a été donné.

`datetime.fold`

In `[0, 1]`. Used to disambiguate wall times during a repeated interval. (A repeated interval occurs when clocks are rolled back at the end of daylight saving time or when the UTC offset for the current zone is decreased for political reasons.) The values 0 and 1 represent, respectively, the earlier and later of the two moments with the same wall time representation.

Ajouté dans la version 3.6.

Opérations gérées :

Opération	Résultat
`datetime2 = datetime1 + timedelta`	(1)
`datetime2 = datetime1 - timedelta`	(2)
`timedelta = datetime1 - datetime2`	(3)
`datetime1 == datetime2` `datetime1 != datetime2`	Equality comparison. (4)

`datetime1 < datetime2`
`datetime1 > datetime2`
`datetime1 <= datetime2`
`datetime1 >= datetime2`
	Order comparison. (5)

(1) `datetime2` is a duration of `timedelta` removed from `datetime1`, moving forward in time if `timedelta.days > 0`, or backward if `timedelta.days < 0`. The result has the same `tzinfo` attribute as the input datetime, and `datetime2 - datetime1 == timedelta` after. `OverflowError` is raised if `datetime2.year` would be smaller than `MINYEAR` or larger than `MAXYEAR`. Note that no time zone adjustments are done even if the input is an aware object.

(2) Computes the `datetime2` such that `datetime2 + timedelta == datetime1`. As for addition, the result has the same `tzinfo` attribute as the input datetime, and no time zone adjustments are done even if the input is aware.

(3) La soustraction d'un *datetime* à un autre *datetime* n'est définie que si les deux opérandes sont naïfs, ou s'ils sont les deux avisés. Si l'un est avisé et que l'autre est naïf, une *TypeError* est levée.

If both are naive, or both are aware and have the same *tzinfo* attribute, the *tzinfo* attributes are ignored, and the result is a *timedelta* object `t` such that `datetime2 + t == datetime1`. No time zone adjustments are done in this case.

If both are aware and have different *tzinfo* attributes, `a-b` acts as if `a` and `b` were first converted to naive UTC datetimes. The result is `(a.replace(tzinfo=None) - a.utcoffset()) - (b.replace(tzinfo=None) - b.utcoffset())` except that the implementation never overflows.

(4) *datetime* objects are equal if they represent the same date and time, taking into account the time zone.

Naive and aware `datetime` objects are never equal.

If both comparands are aware, and have the same `tzinfo` attribute, the `tzinfo` and *fold* attributes are ignored and the base datetimes are compared. If both comparands are aware and have different *tzinfo* attributes, the comparison acts as comparands were first converted to UTC datetimes except that the implementation never overflows. `datetime` instances in a repeated interval are never equal to `datetime` instances in other time zone.

(5) *datetime1* is considered less than *datetime2* when *datetime1* precedes *datetime2* in time, taking into account the time zone.

Order comparison between naive and aware *datetime* objects raises *TypeError*.

If both comparands are aware, and have the same `tzinfo` attribute, the `tzinfo` and *fold* attributes are ignored and the base datetimes are compared. If both comparands are aware and have different *tzinfo* attributes, the comparison acts as comparands were first converted to UTC datetimes except that the implementation never overflows.

Modifié dans la version 3.3 : Les comparaisons d'égalité entre des instances *datetime* naïves et avisées ne lèvent pas d'exception *TypeError*.

Modifié dans la version 3.13 : Comparison between *datetime* object and an instance of the *date* subclass that is not a `datetime` subclass no longer converts the latter to `date`, ignoring the time part and the time zone. The default behavior can be changed by overriding the special comparison methods in subclasses.

Méthodes de l'instance :

`datetime.`**`date`**`()`

Renvoie un objet *date* avec les mêmes année, mois et jour.

`datetime.`**`time`**`()`

Renvoie un objet *time* avec les mêmes heures, minutes, secondes, microsecondes et *fold*. *tzinfo* est `None`. Voir aussi la méthode *timetz()*.

Modifié dans la version 3.6 : La valeur *fold* est copiée vers l'objet *time* renvoyé.

`datetime.`**`timetz`**`()`

Renvoie un objet *time* avec les mêmes attributs heure, minute, seconde, microseconde, *fold* et *tzinfo*. Voir aussi la méthode *time()*.

Modifié dans la version 3.6 : La valeur *fold* est copiée vers l'objet *time* renvoyé.

`datetime.`**`replace`**`(`*year=self.year*, *month=self.month*, *day=self.day*, *hour=self.hour*, *minute=self.minute*, *second=self.second*, *microsecond=self.microsecond*, *tzinfo=self.tzinfo*, *, *fold=0*`)`

Renvoie un *datetime* avec les mêmes attributs, exceptés ceux dont de nouvelles valeurs sont données par les arguments nommés correspondant. Notez que `tzinfo=None` peut être spécifié pour créer un *datetime* naïf depuis un *datetime* avisé sans conversion de la date ou de l'heure.

datetime objects are also supported by generic function *copy.replace()*.

Modifié dans la version 3.6 : Added the *fold* parameter.

`datetime.`**`astimezone`**`(`*tz=None*`)`

Renvoie un objet *datetime* avec un nouvel attribut *tzinfo* valant *tz*, ajustant la date et l'heure pour que le résultat soit le même temps UTC que *self*, mais dans le temps local au fuseau *tz*.

If provided, *tz* must be an instance of a `tzinfo` subclass, and its `utcoffset()` and `dst()` methods must not return `None`. If *self* is naive, it is presumed to represent time in the system time zone.

If called without arguments (or with `tz=None`) the system local time zone is assumed for the target time zone. The `.tzinfo` attribute of the converted datetime instance will be set to an instance of `timezone` with the zone name and offset obtained from the OS.

If `self.tzinfo` is *tz*, `self.astimezone(tz)` is equal to *self* : no adjustment of date or time data is performed. Else the result is local time in the time zone *tz*, representing the same UTC time as *self* : after `astz = dt.astimezone(tz)`, `astz - astz.utcoffset()` will have the same date and time data as `dt - dt.utcoffset()`.

If you merely want to attach a `timezone` object *tz* to a datetime *dt* without adjustment of date and time data, use `dt.replace(tzinfo=tz)`. If you merely want to remove the `timezone` object from an aware datetime *dt* without conversion of date and time data, use `dt.replace(tzinfo=None)`.

Notez que la méthode par défaut `tzinfo.fromutc()` peut être redéfinie dans une sous-classe `tzinfo` pour affecter le résultat renvoyé par `astimezone()`. En ignorant les cas d'erreurs, `astimezone()` se comporte comme :

```
def astimezone(self, tz):
    if self.tzinfo is tz:
        return self
    # Convert self to UTC, and attach the new timezone object.
    utc = (self - self.utcoffset()).replace(tzinfo=tz)
    # Convert from UTC to tz's local time.
    return tz.fromutc(utc)
```

Modifié dans la version 3.3 : *tz* peut maintenant être omis.

Modifié dans la version 3.6 : La méthode `astimezone()` peut maintenant être appelée sur des instances naïves qui sont supposées représenter un temps local au système.

`datetime.`**`utcoffset`**`()`

Si `tzinfo` est `None`, renvoie `None`, sinon renvoie `self.tzinfo.utcoffset(self)`, et lève une exception si l'expression précédente ne renvoie pas `None` ou un objet `timedelta` d'une magnitude inférieure à un jour.

Modifié dans la version 3.7 : Le décalage UTC peut aussi être autre chose qu'un ensemble de minutes.

`datetime.`**`dst`**`()`

Si `tzinfo` est `None`, renvoie `None`, sinon renvoie `self.tzinfo.dst(self)`, et lève une exception si l'expression précédente ne renvoie pas `None` ou un objet `timedelta` d'une magnitude inférieure à un jour.

Modifié dans la version 3.7 : Le décalage DST n'est pas restreint à des minutes entières.

`datetime.`**`tzname`**`()`

Si `tzinfo` est `None`, renvoie `None`, sinon renvoie `self.tzinfo.tzname(self)`, lève une exception si l'expression précédente ne renvoie pas `None` ou une chaîne de caractères,

`datetime.`**`timetuple`**`()`

Renvoie une `time.struct_time` telle que renvoyée par `time.localtime()`.

`d.timetuple()` est équivalent à :

```
time.struct_time((d.year, d.month, d.day,
                  d.hour, d.minute, d.second,
                  d.weekday(), yday, dst))
```

where `yday = d.toordinal() - date(d.year, 1, 1).toordinal() + 1` is the day number within the current year starting with 1 for January 1st. The `tm_isdst` flag of the result is set according to the `dst()` method : `tzinfo` is `None` or `dst()` returns `None`, `tm_isdst` is set to `-1` ; else if `dst()` returns a non-zero value, `tm_isdst` is set to 1 ; else `tm_isdst` is set to 0.

datetime.**utctimetuple**()

If *datetime* instance d is naive, this is the same as d.timetuple() except that *tm_isdst* is forced to 0 regardless of what d.dst() returns. DST is never in effect for a UTC time.

If d is aware, d is normalized to UTC time, by subtracting d.utcoffset(), and a *time.struct_time* for the normalized time is returned. tm_isdst is forced to 0. Note that an *OverflowError* may be raised if d.year was MINYEAR or MAXYEAR and UTC adjustment spills over a year boundary.

Avertissement

Comme les objets datetime naïfs sont traités par de nombreuses méthodes datetime comme des heures locales, il est préférable d'utiliser les datetime avisés pour représenter les heures en UTC ; par conséquent, l'utilisation de datetime.utcfromtimetuple() peut donner des résultats trompeurs. Si vous disposez d'une datetime naïve représentant l'heure UTC, utilisez datetime.replace(tzinfo=timezone.utc) pour la rendre avisée, puis vous pouvez utiliser *datetime.timetuple()*.

datetime.**toordinal**()

Renvoie l'ordinal du calendrier grégorien proleptique de cette date. Identique à self.date().toordinal().

datetime.**timestamp**()

Renvoie l'horodatage *POSIX* correspondant à l'instance *datetime*. La valeur renvoyée est un *float* similaire à ceux renvoyés par *time.time()*.

Naive *datetime* instances are assumed to represent local time and this method relies on the platform C mktime() function to perform the conversion. Since *datetime* supports wider range of values than mktime() on many platforms, this method may raise *OverflowError* or *OSError* for times far in the past or far in the future.

Pour les instances *datetime* avisées, la valeur renvoyée est calculée comme suit :

```
(dt - datetime(1970, 1, 1, tzinfo=timezone.utc)).total_seconds()
```

Ajouté dans la version 3.3.

Modifié dans la version 3.6 : La méthode *timestamp()* utilise l'attribut *fold* pour désambiguïser le temps dans un intervalle répété.

Note

There is no method to obtain the POSIX timestamp directly from a naive *datetime* instance representing UTC time. If your application uses this convention and your system time zone is not set to UTC, you can obtain the POSIX timestamp by supplying tzinfo=timezone.utc :

```
timestamp = dt.replace(tzinfo=timezone.utc).timestamp()
```

ou en calculant l'horodatage (*timestamp* en anglais) directement :

```
timestamp = (dt - datetime(1970, 1, 1)) / timedelta(seconds=1)
```

datetime.**weekday**()

Renvoie le jour de la semaine sous forme de nombre, où lundi vaut 0 et dimanche vaut 6. Identique à self.date().weekday(). Voir aussi *isoweekday()*.

datetime.**isoweekday**()

Renvoie le jour de la semaine sous forme de nombre, où lundi vaut 1 et dimanche vaut 7. Identique à self.date().isoweekday(). Voir aussi *weekday()*, *isocalendar()*.

datetime.**isocalendar**()

Renvoie un *n-uplet nommé* de 3 éléments : year, week et weekday. Identique à self.date().isocalendar().

datetime.**isoformat**(*sep='T'*, *timespec='auto'*)

Renvoie une chaîne représentant la date et l'heure au format ISO 8601 :
— `YYYY-MM-DDTHH:MM:SS.ffffff`, si *microsecond* ne vaut pas 0
— `YYYY-MM-DDTHH:MM:SS`, si *microsecond* vaut 0
Si *utcoffset()* ne renvoie pas `None`, une chaîne est ajoutée, donnant le décalage UTC :
— `YYYY-MM-DDTHH:MM:SS.ffffff+HH:MM[:SS[.ffffff]]`, si *microsecond* ne vaut pas 0
— `YYYY-MM-DDTHH:MM:SS+HH:MM[:SS[.ffffff]]`, si *microsecond* vaut 0
Exemples :

```
>>> from datetime import datetime, timezone
>>> datetime(2019, 5, 18, 15, 17, 8, 132263).isoformat()
'2019-05-18T15:17:08.132263'
>>> datetime(2019, 5, 18, 15, 17, tzinfo=timezone.utc).isoformat()
'2019-05-18T15:17:00+00:00'
```

L'argument optionnel *sep* (par défaut `'T'`) est un séparateur d'un caractère, placé entre les portions du résultat correspondant à la date et à l'heure. Par exemple :

```
>>> from datetime import tzinfo, timedelta, datetime
>>> class TZ(tzinfo):
...     """A time zone with an arbitrary, constant -06:39 offset."""
...     def utcoffset(self, dt):
...         return timedelta(hours=-6, minutes=-39)
...
>>> datetime(2002, 12, 25, tzinfo=TZ()).isoformat(' ')
'2002-12-25 00:00:00-06:39'
>>> datetime(2009, 11, 27, microsecond=100, tzinfo=TZ()).isoformat()
'2009-11-27T00:00:00.000100-06:39'
```

L'argument optionnel *timespec* spécifie le nombre de composants additionnels de temps à inclure (par défaut `'auto'`). Il peut valoir l'une des valeurs suivantes :
— `'auto'` : Identique à `'seconds'` si *microsecond* vaut 0, à `'microseconds'` sinon.
— `'hours'` : Inclut *hour* au format à deux chiffres `HH`.
— `'minutes'` : Inclut *hour* et *minute* au format `HH:MM`.
— `'seconds'` : Inclut *hour*, *minute* et *second* au format `HH:MM:SS`.
— `'milliseconds'` : Inclut le temps complet, mais tronque la partie fractionnaire des millisecondes, au format `HH:MM:SS.sss`.
— `'microseconds'` : Inclut le temps complet, au format `HH:MM:SS.ffffff`.

> **Note**
> Les composants de temps exclus sont tronqués et non arrondis.

Une *ValueError* est levée en cas d'argument *timespec* invalide :

```
>>> from datetime import datetime
>>> datetime.now().isoformat(timespec='minutes')
'2002-12-25T00:00'
>>> dt = datetime(2015, 1, 1, 12, 30, 59, 0)
>>> dt.isoformat(timespec='microseconds')
'2015-01-01T12:30:59.000000'
```

Modifié dans la version 3.6 : Added the *timespec* parameter.

datetime.**__str__**()

For a *datetime* instance `d`, `str(d)` is equivalent to `d.isoformat(' ')`.

datetime.**ctime**()

Renvoie une chaîne représentant la date et l'heure :

```
>>> from datetime import datetime
>>> datetime(2002, 12, 4, 20, 30, 40).ctime()
'Wed Dec  4 20:30:40 2002'
```

La chaîne de caractères en sortie n'inclura *pas* d'informations sur le fuseau horaire, que l'entrée soit avisée ou naïve.

`d.ctime()` est équivalent à :

```
time.ctime(time.mktime(d.timetuple()))
```

sur les plateformes où la fonction C native `ctime()` (que `time.ctime()` invoque, mais pas `datetime.ctime()`) est conforme au standard C.

`datetime.`**`strftime`**(*format*)

Renvoie une chaîne représentant la date et l'heure, contrôlée par une chaîne de format explicite. Voir *strftime() and strptime() Behavior* et `datetime.isoformat()`.

`datetime.`**`__format__`**(*format*)

Identique à `datetime.strftime()`. Cela permet de spécifier une chaîne de format pour un objet `datetime` dans une chaîne de formatage littérale et en utilisant `str.format()`. Voir *strftime() and strptime() Behavior* et `datetime.isoformat()`.

Exemple d'utilisation de la classe `datetime` :

Examples of working with `datetime` objects :

```
>>> from datetime import datetime, date, time, timezone

>>> # Using datetime.combine()
>>> d = date(2005, 7, 14)
>>> t = time(12, 30)
>>> datetime.combine(d, t)
datetime.datetime(2005, 7, 14, 12, 30)

>>> # Using datetime.now()
>>> datetime.now()
datetime.datetime(2007, 12, 6, 16, 29, 43, 79043)   # GMT +1
>>> datetime.now(timezone.utc)
datetime.datetime(2007, 12, 6, 15, 29, 43, 79060, tzinfo=datetime.timezone.utc)

>>> # Using datetime.strptime()
>>> dt = datetime.strptime("21/11/06 16:30", "%d/%m/%y %H:%M")
>>> dt
datetime.datetime(2006, 11, 21, 16, 30)

>>> # Using datetime.timetuple() to get tuple of all attributes
>>> tt = dt.timetuple()
>>> for it in tt:
...     print(it)
...
2006    # year
11      # month
21      # day
16      # hour
30      # minute
0       # second
1       # weekday (0 = Monday)
325     # number of days since 1st January
```

(suite sur la page suivante)

(suite de la page précédente)

```
-1      # dst - method tzinfo.dst() returned None

>>> # Date in ISO format
>>> ic = dt.isocalendar()
>>> for it in ic:
...     print(it)
...
2006    # ISO year
47      # ISO week
2       # ISO weekday

>>> # Formatting a datetime
>>> dt.strftime("%A, %d. %B %Y %I:%M%p")
'Tuesday, 21. November 2006 04:30PM'
>>> 'The {1} is {0:%d}, the {2} is {0:%B}, the {3} is {0:%I:%M%p}.'.format(dt, "day
↪", "month", "time")
'The day is 21, the month is November, the time is 04:30PM.'
```

L'exemple ci-dessous définit une sous-classe `tzinfo` qui regroupe des informations sur les fuseaux horaires pour Kaboul, en Afghanistan, qui a utilisé +4 UTC jusqu'en 1945, puis +4 :30 UTC par la suite :

```
from datetime import timedelta, datetime, tzinfo, timezone

class KabulTz(tzinfo):
    # Kabul used +4 until 1945, when they moved to +4:30
    UTC_MOVE_DATE = datetime(1944, 12, 31, 20, tzinfo=timezone.utc)

    def utcoffset(self, dt):
        if dt.year < 1945:
            return timedelta(hours=4)
        elif (1945, 1, 1, 0, 0) <= dt.timetuple()[:5] < (1945, 1, 1, 0, 30):
            # An ambiguous ("imaginary") half-hour range representing
            # a 'fold' in time due to the shift from +4 to +4:30.
            # If dt falls in the imaginary range, use fold to decide how
            # to resolve. See PEP495.
            return timedelta(hours=4, minutes=(30 if dt.fold else 0))
        else:
            return timedelta(hours=4, minutes=30)

    def fromutc(self, dt):
        # Follow same validations as in datetime.tzinfo
        if not isinstance(dt, datetime):
            raise TypeError("fromutc() requires a datetime argument")
        if dt.tzinfo is not self:
            raise ValueError("dt.tzinfo is not self")

        # A custom implementation is required for fromutc as
        # the input to this function is a datetime with utc values
        # but with a tzinfo set to self.
        # See datetime.astimezone or fromtimestamp.
        if dt.replace(tzinfo=timezone.utc) >= self.UTC_MOVE_DATE:
            return dt + timedelta(hours=4, minutes=30)
        else:
            return dt + timedelta(hours=4)

    def dst(self, dt):
```

(suite sur la page suivante)

(suite de la page précédente)

```
        # Kabul does not observe daylight saving time.
        return timedelta(0)

    def tzname(self, dt):
        if dt >= self.UTC_MOVE_DATE:
            return "+04:30"
        return "+04"
```

Utilisation de `KabulTz` cité plus haut :

```
>>> tz1 = KabulTz()

>>> # Datetime before the change
>>> dt1 = datetime(1900, 11, 21, 16, 30, tzinfo=tz1)
>>> print(dt1.utcoffset())
4:00:00

>>> # Datetime after the change
>>> dt2 = datetime(2006, 6, 14, 13, 0, tzinfo=tz1)
>>> print(dt2.utcoffset())
4:30:00

>>> # Convert datetime to another time zone
>>> dt3 = dt2.astimezone(timezone.utc)
>>> dt3
datetime.datetime(2006, 6, 14, 8, 30, tzinfo=datetime.timezone.utc)
>>> dt2
datetime.datetime(2006, 6, 14, 13, 0, tzinfo=KabulTz())
>>> dt2 == dt3
True
```

8.1.7 Objets `time`

A *time* object represents a (local) time of day, independent of any particular day, and subject to adjustment via a *tzinfo* object.

class datetime.**time**(*hour=0*, *minute=0*, *second=0*, *microsecond=0*, *tzinfo=None*, *, *fold=0*)

Tous les arguments sont optionnels. *tzinfo* peut être `None` ou une instance d'une sous-classe *tzinfo*. Les autres arguments doivent être des nombres entiers, dans les intervalles suivants :

- `0 <= hour < 24`,
- `0 <= minute < 60`,
- `0 <= second < 60`,
- `0 <= microsecond < 1000000`,
- `fold in [0, 1]`.

If an argument outside those ranges is given, *ValueError* is raised. All default to 0 except *tzinfo*, which defaults to `None`.

Attributs de la classe :

time.**min**

Le plus petit objet *time* représentable, `time(0, 0, 0, 0)`.

time.**max**

Le plus grand objet *time* représentable, `time(23, 59, 59, 999999)`.

time.**resolution**

La plus petite différence possible entre deux objets *time* non-égaux, `timedelta(microseconds=1)`, notez cependant que les objets *time* n'implémentent pas d'opérations arithmétiques.

Attributs de l'instance (en lecture seule) :

`time.hour`

Dans `range(24)`.

`time.minute`

Dans `range(60)`.

`time.second`

Dans `range(60)`.

`time.microsecond`

Dans `range(1000000)`.

`time.tzinfo`

L'objet passé comme argument *tzinfo* au constructeur de `time`, ou `None` si aucune valeur n'a été passée.

`time.fold`

In `[0, 1]`. Used to disambiguate wall times during a repeated interval. (A repeated interval occurs when clocks are rolled back at the end of daylight saving time or when the UTC offset for the current zone is decreased for political reasons.) The values 0 and 1 represent, respectively, the earlier and later of the two moments with the same wall time representation.

Ajouté dans la version 3.6.

`time` objects support equality and order comparisons, where `a` is considered less than `b` when `a` precedes `b` in time.

Naive and aware `time` objects are never equal. Order comparison between naive and aware `time` objects raises `TypeError`.

If both comparands are aware, and have the same `tzinfo` attribute, the `tzinfo` and `fold` attributes are ignored and the base times are compared. If both comparands are aware and have different `tzinfo` attributes, the comparands are first adjusted by subtracting their UTC offsets (obtained from `self.utcoffset()`).

Modifié dans la version 3.3 : Equality comparisons between aware and naive `time` instances don't raise `TypeError`.

Dans un contexte booléen, un objet `time` est toujours considéré comme vrai.

Modifié dans la version 3.5 : Avant Python 3.5, un objet `time` était considéré comme faux s'il représentait minuit en UTC. Ce comportement était considéré comme obscur et propice aux erreurs, il a été supprimé en Python 3.5. Voir bpo-13936 pour les détails complets.

Other constructors :

classmethod `time.fromisoformat`(*time_string*)

Renvoie une `time` correspondant à *time_string* dans un format ISO 8601 valide, avec les exceptions suivantes :

1. Les décalages de fuseaux horaires peuvent comporter des fractions de secondes.
2. Le `T` initial, normalement requis dans les cas où il peut y avoir une ambiguïté entre une date et une heure, n'est pas nécessaire.
3. Les fractions de secondes peuvent avoir un nombre quelconque de décimales (tout ce qui dépasse 6 décimales sera tronqué).
4. Les fractions d'heures et de minutes ne sont pas gérées.

Examples :

```
>>> from datetime import time
>>> time.fromisoformat('04:23:01')
datetime.time(4, 23, 1)
>>> time.fromisoformat('T04:23:01')
datetime.time(4, 23, 1)
>>> time.fromisoformat('T042301')
datetime.time(4, 23, 1)
>>> time.fromisoformat('04:23:01.000384')
```

(suite sur la page suivante)

(suite de la page précédente)

```
datetime.time(4, 23, 1, 384)
>>> time.fromisoformat('04:23:01,000384')
datetime.time(4, 23, 1, 384)
>>> time.fromisoformat('04:23:01+04:00')
datetime.time(4, 23, 1, tzinfo=datetime.timezone(datetime.
↪timedelta(seconds=14400)))
>>> time.fromisoformat('04:23:01Z')
datetime.time(4, 23, 1, tzinfo=datetime.timezone.utc)
>>> time.fromisoformat('04:23:01+00:00')
datetime.time(4, 23, 1, tzinfo=datetime.timezone.utc)
```

Ajouté dans la version 3.7.

Modifié dans la version 3.11 : Previously, this method only supported formats that could be emitted by `time.isoformat()`.

classmethod `time.`**`strptime`**(*date_string*, *format*)

Return a `time` corresponding to *date_string*, parsed according to *format*.

If *format* does not contain microseconds or timezone information, this is equivalent to :

```
time(*(time.strptime(date_string, format)[3:6]))
```

`ValueError` is raised if the *date_string* and *format* cannot be parsed by `time.strptime()` or if it returns a value which is not a time tuple. See also *strftime() and strptime() Behavior* and `time.fromisoformat()`.

Ajouté dans la version 3.14.

Méthodes de l'instance :

`time.`**`replace`**(*hour=self.hour*, *minute=self.minute*, *second=self.second*, *microsecond=self.microsecond*, *tzinfo=self.tzinfo*, *, *fold=0*)

Renvoie un objet `time` avec la même valeur, à l'exception des attributs dont les nouvelles valeurs sont spécifiées par les arguments nommés. Notez que `tzinfo=None` peut être spécifié pour créer une instance `time` naïve à partir d'une instance `time` avisée, sans conversion des données de temps.

`time` objects are also supported by generic function `copy.replace()`.

Modifié dans la version 3.6 : Added the *fold* parameter.

`time.`**`isoformat`**(*timespec='auto'*)

Renvoie une chaîne représentant la date au format ISO 8601 :

— `HH:MM:SS.ffffff`, si `microsecond` ne vaut pas 0
— `HH:MM:SS`, si `microsecond` vaut 0
— `HH:MM:SS.ffffff+HH:MM[:SS[.ffffff]]`, si `utcoffset()` ne renvoie pas `None`
— `HH:MM:SS+HH:MM[:SS[.ffffff]]`, si `microsecond` vaut 0 et `utcoffset()` ne renvoie pas `None`

L'argument optionnel *timespec* spécifie le nombre de composants additionnels de temps à inclure (par défaut `'auto'`). Il peut valoir l'une des valeurs suivantes :

— `'auto'` : Identique à `'seconds'` si `microsecond` vaut 0, à `'microseconds'` sinon.
— `'hours'` : Inclut `hour` au format à deux chiffres `HH`.
— `'minutes'` : Inclut `hour` et `minute` au format `HH:MM`.
— `'seconds'` : Inclut `hour`, `minute` et `second` au format `HH:MM:SS`.
— `'milliseconds'` : Inclut le temps complet, mais tronque la partie fractionnaire des millisecondes, au format `HH:MM:SS.sss`.
— `'microseconds'` : Inclut le temps complet, au format `HH:MM:SS.ffffff`.

> **Note**
>
> Les composants de temps exclus sont tronqués et non arrondis.

Une `ValueError` sera levée en cas d'argument *timespec* invalide.

Exemple :

```
>>> from datetime import time
>>> time(hour=12, minute=34, second=56, microsecond=123456).isoformat(timespec=
↪'minutes')
'12:34'
>>> dt = time(hour=12, minute=34, second=56, microsecond=0)
>>> dt.isoformat(timespec='microseconds')
'12:34:56.000000'
>>> dt.isoformat(timespec='auto')
'12:34:56'
```

Modifié dans la version 3.6 : Added the *timespec* parameter.

`time.__str__()`

For a time `t`, `str(t)` is equivalent to `t.isoformat()`.

`time.strftime(format)`

Renvoie une chaîne représentant l'heure, contrôlée par une chaîne de formatage explicite. Voir *strftime() and strptime() Behavior* et `time.isoformat()`.

`time.__format__(format)`

Identique à `time.strftime()`. Cela permet de spécifier une chaîne de formatage pour un objet `time` dans une chaîne de formatage littérale et à l'utilisation de `str.format()`. Voir *strftime() and strptime() Behavior* et `time.isoformat()`.

`time.utcoffset()`

Si `tzinfo` est `None`, renvoie `None`, sinon renvoie `self.tzinfo.utcoffset(None)`, et lève une exception si l'expression précédente ne renvoie pas `None` ou un objet `timedelta` d'une magnitude inférieure à un jour.

Modifié dans la version 3.7 : Le décalage UTC peut aussi être autre chose qu'un ensemble de minutes.

`time.dst()`

Si `tzinfo` est `None`, renvoie `None`, sinon renvoie `self.tzinfo.dst(None)`, et lève une exception si l'expression précédente ne renvoie pas `None` ou un objet `timedelta` d'une magnitude inférieure à un jour.

Modifié dans la version 3.7 : Le décalage DST n'est pas restreint à des minutes entières.

`time.tzname()`

Si `tzinfo` est `None`, renvoie `None`, sinon renvoie `self.tzinfo.tzname(None)`, et lève une exception si l'expression précédente ne renvoie pas `None` ou une chaîne de caractères.

Exemples d'utilisation de `time`

Exemples d'utilisation de l'objet `time` :

```
>>> from datetime import time, tzinfo, timedelta
>>> class TZ1(tzinfo):
...     def utcoffset(self, dt):
...         return timedelta(hours=1)
...     def dst(self, dt):
...         return timedelta(0)
...     def tzname(self,dt):
...         return "+01:00"
...     def  __repr__(self):
...         return f"{self.__class__.__name__}()"
...
>>> t = time(12, 10, 30, tzinfo=TZ1())
>>> t
datetime.time(12, 10, 30, tzinfo=TZ1())
>>> t.isoformat()
```

(suite sur la page suivante)

(suite de la page précédente)

```
'12:10:30+01:00'
>>> t.dst()
datetime.timedelta(0)
>>> t.tzname()
'+01:00'
>>> t.strftime("%H:%M:%S %Z")
'12:10:30 +01:00'
>>> 'The {} is {:%H:%M}.'.format("time", t)
'The time is 12:10.'
```

8.1.8 Objets `tzinfo`

class `datetime.`**`tzinfo`**

Il s'agit d'une classe mère abstraite, ce qui signifie que cette classe ne doit pas être instanciée directement. Définissez une sous-classe de `tzinfo` pour capturer des informations sur un fuseau horaire particulier.

Une instance (d'une sous-classe concrète) de `tzinfo` peut être passée aux constructeurs des objets `datetime` et `time`. Les objets en question voient leurs attributs comme étant en temps local, et l'objet `tzinfo` contient des méthodes pour obtenir le décalage du temps local par rapport à UTC, le nom du fuseau horaire, le décalage d'heure d'été, tous relatifs à un objet de date ou d'heure qui leur est passé.

You need to derive a concrete subclass, and (at least) supply implementations of the standard `tzinfo` methods needed by the `datetime` methods you use. The `datetime` module provides `timezone`, a simple concrete subclass of `tzinfo` which can represent time zones with fixed offset from UTC such as UTC itself or North American EST and EDT.

Special requirement for pickling : A `tzinfo` subclass must have an `__init__()` method that can be called with no arguments, otherwise it can be pickled but possibly not unpickled again. This is a technical requirement that may be relaxed in the future.

A concrete subclass of `tzinfo` may need to implement the following methods. Exactly which methods are needed depends on the uses made of aware `datetime` objects. If in doubt, simply implement all of them.

`tzinfo.`**`utcoffset`**(*dt*)

Renvoie le décalage de l'heure locale par rapport à UTC, sous la forme d'un objet `timedelta` qui est positif à l'est de UTC. Si l'heure locale est à l'ouest de UTC, il doit être négatif.

Cela représente le décalage *total* par rapport à UTC ; par exemple, si un objet `tzinfo` représente à la fois un fuseau horaire et son ajustement à l'heure d'été, `utcoffset()` devrait renvoyer leur somme. Si le décalage UTC n'est pas connu, elle renvoie `None`. Sinon, la valeur renvoyée doit être un objet `timedelta` compris strictement entre `-timedelta(hours=24)` et `timedelta(hours=24)` (l'amplitude du décalage doit être inférieure à un jour). La plupart des implémentations de `utcoffset()` ressembleront probablement à l'une des deux suivantes :

```
return CONSTANT                 # fixed-offset class
return CONSTANT + self.dst(dt)  # daylight-aware class
```

Si `utcoffset()` ne renvoie pas `None`, `dst()` ne doit pas non plus renvoyer `None`.

L'implémentation par défaut de `utcoffset()` lève une `NotImplementedError`.

Modifié dans la version 3.7 : Le décalage UTC peut aussi être autre chose qu'un ensemble de minutes.

`tzinfo.`**`dst`**(*dt*)

Renvoie le réglage de l'heure d'été (DST), sous la forme d'un objet `timedelta` ou `None` si l'information DST n'est pas connue.

Return `timedelta(0)` if DST is not in effect. If DST is in effect, return the offset as a `timedelta` object (see `utcoffset()` for details). Note that DST offset, if applicable, has already been added to the UTC offset returned by `utcoffset()`, so there's no need to consult `dst()` unless you're interested in obtaining DST info separately. For example, `datetime.timetuple()` calls its `tzinfo` attribute's `dst()` method to determine

how the `tm_isdst` flag should be set, and `tzinfo.fromutc()` calls `dst()` to account for DST changes when crossing time zones.

Une instance *tz* d'une sous-classe `tzinfo` convenant à la fois pour une heure standard et une heure d'été doit être cohérente :

```
tz.utcoffset(dt) - tz.dst(dt)
```

must return the same result for every `datetime` *dt* with `dt.tzinfo == tz`. For sane `tzinfo` subclasses, this expression yields the time zone's "standard offset", which should not depend on the date or the time, but only on geographic location. The implementation of `datetime.astimezone()` relies on this, but cannot detect violations ; it's the programmer's responsibility to ensure it. If a `tzinfo` subclass cannot guarantee this, it may be able to override the default implementation of `tzinfo.fromutc()` to work correctly with `astimezone()` regardless.

La plupart des implémentations de `dst()` ressembleront probablement à l'une des deux suivantes :

```
def dst(self, dt):
    # a fixed-offset class:  doesn't account for DST
    return timedelta(0)
```

ou :

```
def dst(self, dt):
    # Code to set dston and dstoff to the time zone's DST
    # transition times based on the input dt.year, and expressed
    # in standard local time.

    if dston <= dt.replace(tzinfo=None) < dstoff:
        return timedelta(hours=1)
    else:
        return timedelta(0)
```

L'implémentation par défaut de `dst()` lève une `NotImplementedError`.

Modifié dans la version 3.7 : Le décalage DST n'est pas restreint à des minutes entières.

`tzinfo.tzname(dt)`

Return the time zone name corresponding to the `datetime` object *dt*, as a string. Nothing about string names is defined by the `datetime` module, and there's no requirement that it mean anything in particular. For example, `"GMT"`, `"UTC"`, `"-500"`, `"-5:00"`, `"EDT"`, `"US/Eastern"`, `"America/New York"` are all valid replies. Return `None` if a string name isn't known. Note that this is a method rather than a fixed string primarily because some `tzinfo` subclasses will wish to return different names depending on the specific value of *dt* passed, especially if the `tzinfo` class is accounting for daylight time.

L'implémentation par défaut de `tzname()` lève une `NotImplementedError`.

Ces méthodes sont appelées par les objets `datetime` et `time`, en réponse à leurs méthodes de mêmes noms. Un objet `datetime` se passe lui-même en tant qu'argument, et un objet `time` passe `None`. Les méthodes des sous-classes `tzinfo` doivent alors être prêtes à recevoir un argument `None` pour *dt*, ou une instance de `datetime`.

Quand `None` est passé, il est de la responsabilité du *designer* de la classe de choisir la meilleure réponse. Par exemple, renvoyer `None` est approprié si la classe souhaite signaler que les objets de temps ne participent pas au protocole `tzinfo`. Il peut être plus utile pour `utcoffset(None)` de renvoyer le décalage UTC standard, comme il n'existe aucune autre convention pour obtenir ce décalage.

When a `datetime` object is passed in response to a `datetime` method, `dt.tzinfo` is the same object as *self*. `tzinfo` methods can rely on this, unless user code calls `tzinfo` methods directly. The intent is that the `tzinfo` methods interpret *dt* as being in local time, and not need worry about objects in other time zones.

Il y a une dernière méthode de `tzinfo` que les sous-classes peuvent vouloir redéfinir :

`tzinfo.fromutc(dt)`

This is called from the default `datetime.astimezone()` implementation. When called from that, `dt.`

`tzinfo` is *self*, and *dt*'s date and time data are to be viewed as expressing a UTC time. The purpose of `fromutc()` is to adjust the date and time data, returning an equivalent datetime in *self*'s local time.

Most `tzinfo` subclasses should be able to inherit the default `fromutc()` implementation without problems. It's strong enough to handle fixed-offset time zones, and time zones accounting for both standard and daylight time, and the latter even if the DST transition times differ in different years. An example of a time zone the default `fromutc()` implementation may not handle correctly in all cases is one where the standard offset (from UTC) depends on the specific date and time passed, which can happen for political reasons. The default implementations of `astimezone()` and `fromutc()` may not produce the result you want if the result is one of the hours straddling the moment the standard offset changes.

En omettant le code des cas d'erreurs, l'implémentation par défaut de `fromutc()` se comporte comme suit :

```
def fromutc(self, dt):
    # raise ValueError error if dt.tzinfo is not self
    dtoff = dt.utcoffset()
    dtdst = dt.dst()
    # raise ValueError if dtoff is None or dtdst is None
    delta = dtoff - dtdst  # this is self's standard offset
    if delta:
        dt += delta   # convert to standard local time
        dtdst = dt.dst()
        # raise ValueError if dtdst is None
    if dtdst:
        return dt + dtdst
    else:
        return dt
```

Dans le fichier `tzinfo_examples.py` il y a des exemples de `tzinfo` classes :

```
from datetime import tzinfo, timedelta, datetime

ZERO = timedelta(0)
HOUR = timedelta(hours=1)
SECOND = timedelta(seconds=1)

# A class capturing the platform's idea of local time.
# (May result in wrong values on historical times in
#  timezones where UTC offset and/or the DST rules had
#  changed in the past.)
import time as _time

STDOFFSET = timedelta(seconds = -_time.timezone)
if _time.daylight:
    DSTOFFSET = timedelta(seconds = -_time.altzone)
else:
    DSTOFFSET = STDOFFSET

DSTDIFF = DSTOFFSET - STDOFFSET

class LocalTimezone(tzinfo):

    def fromutc(self, dt):
        assert dt.tzinfo is self
        stamp = (dt - datetime(1970, 1, 1, tzinfo=self)) // SECOND
        args = _time.localtime(stamp)[:6]
        dst_diff = DSTDIFF // SECOND
        # Detect fold
        fold = (args == _time.localtime(stamp - dst_diff))
```

(suite sur la page suivante)

(suite de la page précédente)

```
        return datetime(*args, microsecond=dt.microsecond,
                        tzinfo=self, fold=fold)

    def utcoffset(self, dt):
        if self._isdst(dt):
            return DSTOFFSET
        else:
            return STDOFFSET

    def dst(self, dt):
        if self._isdst(dt):
            return DSTDIFF
        else:
            return ZERO

    def tzname(self, dt):
        return _time.tzname[self._isdst(dt)]

    def _isdst(self, dt):
        tt = (dt.year, dt.month, dt.day,
              dt.hour, dt.minute, dt.second,
              dt.weekday(), 0, 0)
        stamp = _time.mktime(tt)
        tt = _time.localtime(stamp)
        return tt.tm_isdst > 0

Local = LocalTimezone()


# A complete implementation of current DST rules for major US time zones.

def first_sunday_on_or_after(dt):
    days_to_go = 6 - dt.weekday()
    if days_to_go:
        dt += timedelta(days_to_go)
    return dt


# US DST Rules
#
# This is a simplified (i.e., wrong for a few cases) set of rules for US
# DST start and end times. For a complete and up-to-date set of DST rules
# and timezone definitions, visit the Olson Database (or try pytz):
# http://www.twinsun.com/tz/tz-link.htm
# https://sourceforge.net/projects/pytz/ (might not be up-to-date)
#
# In the US, since 2007, DST starts at 2am (standard time) on the second
# Sunday in March, which is the first Sunday on or after Mar 8.
DSTSTART_2007 = datetime(1, 3, 8, 2)
# and ends at 2am (DST time) on the first Sunday of Nov.
DSTEND_2007 = datetime(1, 11, 1, 2)
# From 1987 to 2006, DST used to start at 2am (standard time) on the first
# Sunday in April and to end at 2am (DST time) on the last
# Sunday of October, which is the first Sunday on or after Oct 25.
DSTSTART_1987_2006 = datetime(1, 4, 1, 2)
DSTEND_1987_2006 = datetime(1, 10, 25, 2)
```

(suite sur la page suivante)

(suite de la page précédente)

```
# From 1967 to 1986, DST used to start at 2am (standard time) on the last
# Sunday in April (the one on or after April 24) and to end at 2am (DST time)
# on the last Sunday of October, which is the first Sunday
# on or after Oct 25.
DSTSTART_1967_1986 = datetime(1, 4, 24, 2)
DSTEND_1967_1986 = DSTEND_1987_2006

def us_dst_range(year):
    # Find start and end times for US DST. For years before 1967, return
    # start = end for no DST.
    if 2006 < year:
        dststart, dstend = DSTSTART_2007, DSTEND_2007
    elif 1986 < year < 2007:
        dststart, dstend = DSTSTART_1987_2006, DSTEND_1987_2006
    elif 1966 < year < 1987:
        dststart, dstend = DSTSTART_1967_1986, DSTEND_1967_1986
    else:
        return (datetime(year, 1, 1), ) * 2

    start = first_sunday_on_or_after(dststart.replace(year=year))
    end = first_sunday_on_or_after(dstend.replace(year=year))
    return start, end


class USTimeZone(tzinfo):

    def __init__(self, hours, reprname, stdname, dstname):
        self.stdoffset = timedelta(hours=hours)
        self.reprname = reprname
        self.stdname = stdname
        self.dstname = dstname

    def __repr__(self):
        return self.reprname

    def tzname(self, dt):
        if self.dst(dt):
            return self.dstname
        else:
            return self.stdname

    def utcoffset(self, dt):
        return self.stdoffset + self.dst(dt)

    def dst(self, dt):
        if dt is None or dt.tzinfo is None:
            # An exception may be sensible here, in one or both cases.
            # It depends on how you want to treat them.  The default
            # fromutc() implementation (called by the default astimezone()
            # implementation) passes a datetime with dt.tzinfo is self.
            return ZERO
        assert dt.tzinfo is self
        start, end = us_dst_range(dt.year)
        # Can't compare naive to aware objects, so strip the timezone from
        # dt first.
        dt = dt.replace(tzinfo=None)
```

(suite sur la page suivante)

(suite de la page précédente)

```
        if start + HOUR <= dt < end - HOUR:
            # DST is in effect.
            return HOUR
        if end - HOUR <= dt < end:
            # Fold (an ambiguous hour): use dt.fold to disambiguate.
            return ZERO if dt.fold else HOUR
        if start <= dt < start + HOUR:
            # Gap (a non-existent hour): reverse the fold rule.
            return HOUR if dt.fold else ZERO
        # DST is off.
        return ZERO

    def fromutc(self, dt):
        assert dt.tzinfo is self
        start, end = us_dst_range(dt.year)
        start = start.replace(tzinfo=self)
        end = end.replace(tzinfo=self)
        std_time = dt + self.stdoffset
        dst_time = std_time + HOUR
        if end <= dst_time < end + HOUR:
            # Repeated hour
            return std_time.replace(fold=1)
        if std_time < start or dst_time >= end:
            # Standard time
            return std_time
        if start <= std_time < end - HOUR:
            # Daylight saving time
            return dst_time


Eastern  = USTimeZone(-5, "Eastern",  "EST", "EDT")
Central  = USTimeZone(-6, "Central",  "CST", "CDT")
Mountain = USTimeZone(-7, "Mountain", "MST", "MDT")
Pacific  = USTimeZone(-8, "Pacific",  "PST", "PDT")
```

Notez que, deux fois par an, on rencontre des subtilités inévitables dans les sous-classes de `tzinfo` représentant à la fois des heures standard et d'été, au passage de l'une à l'autre. Concrètement, considérez le fuseau de l'est des États-Unis (UTC -0500), où EDT (heure d'été) débute à la minute qui suit 1 :59 (EST) le second dimanche de mars, et se termine à la minute qui suit 1 :59 (EDT) le premier dimanche de novembre :

```
  UTC    3:MM  4:MM  5:MM  6:MM  7:MM  8:MM
  EST   22:MM 23:MM  0:MM  1:MM  2:MM  3:MM
  EDT   23:MM  0:MM  1:MM  2:MM  3:MM  4:MM

start   22:MM 23:MM  0:MM  1:MM  3:MM  4:MM

  end   23:MM  0:MM  1:MM  1:MM  2:MM  3:MM
```

Quand l'heure d'été débute (la ligne « *start* »), l'horloge locale passe de 1 :59 à 3 :00. Une heure de la forme 2 :MM n'a pas vraiment de sens ce jour là, donc `astimezone(Eastern)` ne fournira pas de résultat avec `hour == 2` pour le jour où débute l'heure d'été. Par exemple, lors du changement d'heure du printemps 2016, nous obtenons :

```
>>> from datetime import datetime, timezone
>>> from tzinfo_examples import HOUR, Eastern
>>> u0 = datetime(2016, 3, 13, 5, tzinfo=timezone.utc)
>>> for i in range(4):
...     u = u0 + i*HOUR
```

(suite sur la page suivante)

(suite de la page précédente)

```
...     t = u.astimezone(Eastern)
...     print(u.time(), 'UTC =', t.time(), t.tzname())
...
05:00:00 UTC = 00:00:00 EST
06:00:00 UTC = 01:00:00 EST
07:00:00 UTC = 03:00:00 EDT
08:00:00 UTC = 04:00:00 EDT
```

When DST ends (the "end" line), there's a potentially worse problem : there's an hour that can't be spelled unambiguously in local wall time : the last hour of daylight time. In Eastern, that's times of the form 5 :MM UTC on the day daylight time ends. The local wall clock leaps from 1 :59 (daylight time) back to 1 :00 (standard time) again. Local times of the form 1 :MM are ambiguous. `astimezone()` mimics the local clock's behavior by mapping two adjacent UTC hours into the same local hour then. In the Eastern example, UTC times of the form 5 :MM and 6 :MM both map to 1 :MM when converted to Eastern, but earlier times have the `fold` attribute set to 0 and the later times have it set to 1. For example, at the Fall back transition of 2016, we get :

```
>>> u0 = datetime(2016, 11, 6, 4, tzinfo=timezone.utc)
>>> for i in range(4):
...     u = u0 + i*HOUR
...     t = u.astimezone(Eastern)
...     print(u.time(), 'UTC =', t.time(), t.tzname(), t.fold)
...
04:00:00 UTC = 00:00:00 EDT 0
05:00:00 UTC = 01:00:00 EDT 0
06:00:00 UTC = 01:00:00 EST 1
07:00:00 UTC = 02:00:00 EST 0
```

Note that the `datetime` instances that differ only by the value of the `fold` attribute are considered equal in comparisons.

Applications that can't bear wall-time ambiguities should explicitly check the value of the `fold` attribute or avoid using hybrid `tzinfo` subclasses ; there are no ambiguities when using `timezone`, or any other fixed-offset `tzinfo` subclass (such as a class representing only EST (fixed offset -5 hours), or only EDT (fixed offset -4 hours)).

Voir aussi

zoneinfo
The `datetime` module has a basic `timezone` class (for handling arbitrary fixed offsets from UTC) and its `timezone.utc` attribute (a UTC `timezone` instance).

`zoneinfo` brings the *IANA time zone database* (also known as the Olson database) to Python, and its usage is recommended.

IANA time zone database
La *Time Zone Database* (souvent appelée *tz*, *tzdata* ou *zoneinfo*) contient les codes et les données représentant l'historique du temps local pour un grand nombre d'emplacements représentatifs autour du globe. Elle est mise à jour périodiquement, pour refléter les changements opérés par des politiques sur les bornes du fuseau, les décalages UTC, et les règles de passage à l'heure d'été.

8.1.9 Objets `timezone`

The `timezone` class is a subclass of `tzinfo`, each instance of which represents a time zone defined by a fixed offset from UTC.

Objects of this class cannot be used to represent time zone information in the locations where different offsets are used in different days of the year or where historical changes have been made to civil time.

class datetime.**timezone**(*offset*, *name=None*)

L'argument *offset* doit être spécifié comme un objet `timedelta` représentant la différence entre le temps local et UTC. Il doit être strictement compris entre `-timedelta(hours=24)` et `timedelta(hours=24)`, autrement une `ValueError` est levée.

L'argument *name* est optionnel. S'il est spécifié, il doit être une chaîne qui sera utilisée comme valeur de retour de la méthode `datetime.tzname()`.

Ajouté dans la version 3.2.

Modifié dans la version 3.7 : Le décalage UTC peut aussi être autre chose qu'un ensemble de minutes.

timezone.**utcoffset**(*dt*)

Renvoie la valeur fixe spécifiée lorsque l'instance `timezone` est construite.

L'argument *dt* est ignoré. La valeur de retour est une instance `timedelta` égale à la différence entre le temps local et UTC.

Modifié dans la version 3.7 : Le décalage UTC peut aussi être autre chose qu'un ensemble de minutes.

timezone.**tzname**(*dt*)

Renvoie la valeur fixe spécifiée lorsque l'instance `timezone` est construite.

Si *name* n'est pas fourni au constructeur, le nom renvoyé par `tzname(dt)` est généré comme suit à partir de la valeur de `offset`. Si *offset* vaut `timedelta(0)`, le nom sera « UTC », autrement le nom sera une chaîne de la forme `UTC±HH:MM`, où ± est le signe d'`offset`, et HH et MM sont respectivement les représentations à deux chiffres de `offset.hours` et `offset.minutes`.

Modifié dans la version 3.6 : Le nom généré à partir de `offset=timedelta(0)` est maintenant `'UTC'` plutôt que `'UTC+00:00'`.

timezone.**dst**(*dt*)

Renvoie toujours `None`.

timezone.**fromutc**(*dt*)

Renvoie `dt + offset`. L'argument *dt* doit être une instance avisée de `datetime`, avec `tzinfo` valant `self`.

Attributs de la classe :

timezone.**utc**

The UTC time zone, `timezone(timedelta(0))`.

8.1.10 `strftime()` and `strptime()` Behavior

Les objets `date`, `datetime` et `time` comportent tous une méthode `strftime(format)`, pour créer une représentation du temps sous forme d'une chaîne, contrôlée par une chaîne de formatage explicite.

Conversely, the `date.strptime()`, `datetime.strptime()` and `time.strptime()` class methods create an object from a string representing the time and a corresponding format string.

The table below provides a high-level comparison of `strftime()` versus `strptime()` :

	`strftime`	`strptime`
Utilisation	Convertit un objet en une chaîne selon un format donné	Parse a string into an object given a corresponding format
Type de méthode	Méthode d'instance	Méthode de classe
Signature	`strftime(format)`	`strptime(date_string, format)`

`strftime()` and `strptime()` Format Codes

Ces méthodes acceptent des codes de formatage qui peuvent être utilisés pour analyser et formater les dates :

```
>>> datetime.strptime('31/01/22 23:59:59.999999',
...                   '%d/%m/%y %H:%M:%S.%f')
datetime.datetime(2022, 1, 31, 23, 59, 59, 999999)
>>> _.strftime('%a %d %b %Y, %I:%M%p')
'Mon 31 Jan 2022, 11:59PM'
```

Voici la liste de tous les codes de formatage requis par le C standard 1989, et ils fonctionnent sur toutes les plateformes possédant une implémentation de C standard.

Directive	Signification	Exemple	Notes
`%a`	Jour de la semaine abrégé dans la langue locale.	Sun, Mon, ..., Sat (en_US) ; Lu, Ma, ..., Di (*fr_FR*)	(1)
`%A`	Jour de la semaine complet dans la langue locale.	*Sunday*, *Monday*, ..., *Saturday* (*en_US*) ; Lundi, Mardi, ..., Dimanche (*fr_FR*)	(1)
`%w`	Jour de la semaine en chiffre, avec 0 pour le dimanche et 6 pour le samedi.	0, 1, ..., 6	
`%d`	Jour du mois sur deux chiffres.	01, 02, ..., 31	(9)
`%b`	Nom du mois abrégé dans la langue locale.	Jan, Feb, ..., Dec (*en_US*) ; janv., févr., ..., déc. (*fr_FR*)	(1)
`%B`	Nom complet du mois dans la langue locale.	*January*, *February*, ..., *December* (*en_US*) ; janvier, février, ..., décembre (*fr_FR*)	(1)
`%m`	Numéro du mois sur deux chiffres.	01, 02, ..., 12	(9)
`%y`	Année sur deux chiffres (sans le siècle).	00, 01, ..., 99	(9)
`%Y`	Année complète sur quatre chiffres.	0001, 0002, ..., 2013, 2014, ..., 9998, 9999	(2)
`%H`	Heure à deux chiffres de 00 à 23.	00, 01, ..., 23	(9)
`%I`	Heure à deux chiffres pour les horloges 12h (01 à 12).	01, 02, ..., 12	(9)
`%p`	Équivalent local à AM/PM.	AM, PM (en_US) ; am, pm (de_DE)	(1), (3)
`%M`	Minutes sur deux chiffres.	00, 01, ..., 59	(9)
`%S`	Secondes sur deux chiffres.	00, 01, ..., 59	(4), (9)
`%f`	Microsecondes sur 6 chiffres.	000000, 000001, ..., 999999	(5)
`%z`	Décalage horaire UTC sous la forme `±HHMM[SS[.ffffff]]` (chaîne vide si l'instance est naïve).	(vide), +0000, -0400, +1030, +063415, -030712.345216	(6)
`%Z`	Nom du fuseau horaire (chaîne vide si l'instance est naïve).	(vide), UTC, GMT	(6)
`%j`	Numéro du jour dans l'an-	001, 002, ..., 366	(9)

Plusieurs directives supplémentaires non requises par la norme C89 sont incluses pour des raisons de commodité. Ces paramètres correspondent tous aux valeurs de date de la norme ISO 8601.

Directive	Signification	Exemple	Notes
`%G`	Année complète ISO 8601 représentant l'année contenant la plus grande partie de la semaine ISO (`%V`).	0001, 0002, ..., 2013, 2014, ..., 9998, 9999	(8)
`%u`	Jour de la semaine ISO 8601 où 1 correspond au lundi.	1, 2, ..., 7	
`%V`	Numéro de la semaine ISO 8601, avec lundi étant le premier jour de la semaine. La semaine 01 est la semaine contenant le 4 janvier.	01, 02, ..., 53	(8), (9)
`%:z`	UTC offset in the form `±HH:MM[:SS[.ffffff]]` (empty string if the object is naive).	(empty), +00 :00, -04 :00, +10 :30, +06 :34 :15, -03 :07 :12.345216	(6)

These may not be available on all platforms when used with the *`strftime()`* method. The ISO 8601 year and ISO 8601 week directives are not interchangeable with the year and week number directives above. Calling *`strptime()`* with incomplete or ambiguous ISO 8601 directives will raise a *`ValueError`*.

The full set of format codes supported varies across platforms, because Python calls the platform C library's `strftime()` function, and platform variations are common. To see the full set of format codes supported on your platform, consult the *`strftime(3)`* documentation. There are also differences between platforms in handling of unsupported format specifiers.

Ajouté dans la version 3.6 : `%G`, `%u` et `%V` ont été ajoutés.

Ajouté dans la version 3.12 : `%:z` was added.

Détail technique

Broadly speaking, `d.strftime(fmt)` acts like the *`time`* module's `time.strftime(fmt, d.timetuple())` although not all objects support a *`timetuple()`* method.

For the *`datetime.strptime()`* class method, the default value is `1900-01-01T00:00:00.000` : any components not specified in the format string will be pulled from the default value. [4]

L'utilisation de `datetime.strptime(date_string, format)` équivaut à :

```
datetime(*(time.strptime(date_string, format)[0:6]))
```

except when the format includes sub-second components or time zone offset information, which are supported in `datetime.strptime` but are discarded by `time.strptime`.

For *`time`* objects, the format codes for year, month, and day should not be used, as `time` objects have no such values. If they're used anyway, 1900 is substituted for the year, and 1 for the month and day.

For *`date`* objects, the format codes for hours, minutes, seconds, and microseconds should not be used, as *`date`* objects have no such values. If they're used anyway, 0 is substituted for them.

Pour la même raison, la gestion des chaînes contenant des caractères (ou points) Unicode qui ne peuvent pas être représentés dans la *locale* actuelle dépend aussi de la plateforme. Sur certaines plateformes, ces caractères sont conservés tels quels dans la sortie, alors que sur d'autres plateformes `strftime` lève une *`UnicodeError`* ou renvoie une chaîne vide.

Notes :

(1) Comme le format dépend de la locale courante, les assomptions sur la valeur de retour doivent être prises soigneusement. L'ordre des champs variera (par exemple, « mois/jour/année » au lieu de « année/mois/jour »), et le résultat peut contenir des caractères non-ASCII.

4. Passing `datetime.strptime('Feb 29', '%b %d')` will fail since 1900 is not a leap year.

(2) The *strptime()* method can parse years in the full [1, 9999] range, but years < 1000 must be zero-filled to 4-digit width.

Modifié dans la version 3.2 : In previous versions, *strftime()* method was restricted to years >= 1900.

Modifié dans la version 3.3 : In version 3.2, *strftime()* method was restricted to years >= 1000.

(3) When used with the *strptime()* method, the `%p` directive only affects the output hour field if the `%I` directive is used to parse the hour.

(4) Unlike the *time* module, the `datetime` module does not support leap seconds.

(5) When used with the *strptime()* method, the `%f` directive accepts from one to six digits and zero pads on the right. `%f` is an extension to the set of format characters in the C standard (but implemented separately in datetime objects, and therefore always available).

(6) For a naive object, the `%z`, `%:z` and `%Z` format codes are replaced by empty strings.

Pour un objet avisé :

%z

utcoffset() is transformed into a string of the form `±HHMM[SS[.ffffff]]`, where `HH` is a 2-digit string giving the number of UTC offset hours, `MM` is a 2-digit string giving the number of UTC offset minutes, `SS` is a 2-digit string giving the number of UTC offset seconds and `ffffff` is a 6-digit string giving the number of UTC offset microseconds. The `ffffff` part is omitted when the offset is a whole number of seconds and both the `ffffff` and the `SS` part is omitted when the offset is a whole number of minutes. For example, if *utcoffset()* returns `timedelta(hours=-3, minutes=-30)`, `%z` is replaced with the string `'-0330'`.

Modifié dans la version 3.7 : Le décalage UTC peut aussi être autre chose qu'un ensemble de minutes.

Modifié dans la version 3.7 : When the `%z` directive is provided to the *strptime()* method, the UTC offsets can have a colon as a separator between hours, minutes and seconds. For example, `'+01:00:00'` will be parsed as an offset of one hour. In addition, providing `'Z'` is identical to `'+00:00'`.

%:z

Behaves exactly as `%z`, but has a colon separator added between hours, minutes and seconds.

%Z

In *strftime()*, `%Z` is replaced by an empty string if *tzname()* returns `None` ; otherwise `%Z` is replaced by the returned value, which must be a string.

strptime() only accepts certain values for `%Z` :

1. toute valeur dans `time.tzname` pour votre machine locale
2. les valeurs `UTC` et `GMT` codés en dur

Ainsi, quelqu'un qui vit au Japon peut avoir comme valeurs valides `JST`, `UTC` et `GMT`, mais probablement pas `EST`. Les valeurs invalides lèvent `ValueError`.

Modifié dans la version 3.2 : When the `%z` directive is provided to the *strptime()* method, an aware *datetime* object will be produced. The `tzinfo` of the result will be set to a *timezone* instance.

(7) When used with the *strptime()* method, `%U` and `%W` are only used in calculations when the day of the week and the calendar year (`%Y`) are specified.

(8) Similar to `%U` and `%W`, `%V` is only used in calculations when the day of the week and the ISO year (`%G`) are specified in a *strptime()* format string. Also note that `%G` and `%Y` are not interchangeable.

(9) When used with the *strptime()* method, the leading zero is optional for formats `%d`, `%m`, `%H`, `%I`, `%M`, `%S`, `%j`, `%U`, `%W`, and `%V`. Format `%y` does require a leading zero.

(10) When parsing a month and day using *strptime()*, always include a year in the format. If the value you need to parse lacks a year, append an explicit dummy leap year. Otherwise your code will raise an exception when it encounters leap day because the default year used by the parser is not a leap year. Users run into this bug every four years...

```
>>> month_day = "02/29"
>>> datetime.strptime(f"{month_day};1984", "%m/%d;%Y")  # No leap year bug.
datetime.datetime(1984, 2, 29, 0, 0)
```

Deprecated since version 3.13, will be removed in version 3.15 : *strptime()* calls using a format string containing a day of month without a year now emit a *DeprecationWarning*. In 3.15 or later we may change this into an error or change the default year to a leap year. See gh-70647.

Notes

8.2 zoneinfo --- IANA time zone support

Ajouté dans la version 3.9.

Code source : Lib/zoneinfo

The *zoneinfo* module provides a concrete time zone implementation to support the IANA time zone database as originally specified in **PEP 615**. By default, *zoneinfo* uses the system's time zone data if available ; if no system time zone data is available, the library will fall back to using the first-party tzdata package available on PyPI.

> **Voir aussi**
>
> **Module : *datetime***
> Fournit les types *time* et *datetime* attendus par *ZoneInfo*.
>
> **Package tzdata**
> Paquet « quasi natif » maintenu par les développeurs de CPython pour fournir les données des fuseaux horaires via PyPI.

Availability : not WASI.

This module does not work or is not available on WebAssembly. See *Plateformes WebAssembly* for more information.

8.2.1 Utilisation de ZoneInfo

ZoneInfo est une implémentation concrète de la classe de base abstraite *datetime.tzinfo* et est destinée à être rattachée à `tzinfo`, par le constructeur, par *datetime.replace* ou par *datetime.astimezone* :

```
>>> from zoneinfo import ZoneInfo
>>> from datetime import datetime, timedelta

>>> dt = datetime(2020, 10, 31, 12, tzinfo=ZoneInfo("America/Los_Angeles"))
>>> print(dt)
2020-10-31 12:00:00-07:00

>>> dt.tzname()
'PDT'
```

Les *datetime* construits de cette manière sont compatibles avec l'arithmétique *datetime* et gèrent le passage à l'heure d'été sans autre intervention :

```
>>> dt_add = dt + timedelta(days=1)

>>> print(dt_add)
2020-11-01 12:00:00-08:00

>>> dt_add.tzname()
'PST'
```

Ces fuseaux horaires prennent aussi en charge l'attribut `fold` introduit dans la **PEP 495**. Pendant les transitions des décalages horaires qui induisent des temps ambigus (comme le passage de l'heure d'été à l'heure normale), le décalage *avant* la transition est utilisé quand `fold=0`, et le décalage *après* la transition est utilisé quand `fold=1`, par exemple :

```
>>> dt = datetime(2020, 11, 1, 1, tzinfo=ZoneInfo("America/Los_Angeles"))
>>> print(dt)
```

(suite sur la page suivante)

(suite de la page précédente)

```
2020-11-01 01:00:00-07:00

>>> print(dt.replace(fold=1))
2020-11-01 01:00:00-08:00
```

Lors de la conversion à partir d'un autre fuseau horaire, *fold* est réglé à la valeur correcte :

```
>>> from datetime import timezone
>>> LOS_ANGELES = ZoneInfo("America/Los_Angeles")
>>> dt_utc = datetime(2020, 11, 1, 8, tzinfo=timezone.utc)

>>> # Before the PDT -> PST transition
>>> print(dt_utc.astimezone(LOS_ANGELES))
2020-11-01 01:00:00-07:00

>>> # After the PDT -> PST transition
>>> print((dt_utc + timedelta(hours=1)).astimezone(LOS_ANGELES))
2020-11-01 01:00:00-08:00
```

8.2.2 Sources de données

The `zoneinfo` module does not directly provide time zone data, and instead pulls time zone information from the system time zone database or the first-party PyPI package tzdata, if available. Some systems, including notably Windows systems, do not have an IANA database available, and so for projects targeting cross-platform compatibility that require time zone data, it is recommended to declare a dependency on tzdata. If neither system data nor tzdata are available, all calls to `ZoneInfo` will raise `ZoneInfoNotFoundError`.

Configurer les sources de données

Lorsque `ZoneInfo(key)` est appelé, le constructeur recherche d'abord un fichier nommé `key` dans les répertoires spécifiés par `TZPATH` et, en cas d'échec, recherche dans le paquet *tzdata*. Ce comportement peut être configuré de trois manières :

1. La valeur par défaut `TZPATH`, lorsqu'elle n'est pas spécifiée autrement, peut être configurée à *la compilation*.
2. `TZPATH` peut être configuré en utilisant *une variable d'environnement*.
3. À *l'exécution*, le chemin de recherche peut être manipulé à l'aide de la fonction `reset_tzpath()`.

Configuration à la compilation

Par défaut, `TZPATH` indique plusieurs emplacements de déploiement courants pour la base de données de fuseaux horaires (sauf sous Windows, où il n'existe pas de consensus pour l'emplacement des données de fuseaux horaires). Sur les systèmes POSIX, les distributeurs en aval ainsi que ceux qui construisent Python à partir des sources et qui savent où sont déployées les données de fuseau horaire de leur système peuvent modifier le chemin par défaut en spécifiant l'option de compilation `TZPATH` (ou, plus probablement, en utilisant l'option `--with-tzpath` du script `configure`), qui doit être une chaîne délimitée par `os.pathsep`.

Sur toutes les plates-formes, la valeur configurée est disponible en tant que clé de `TZPATH` dans `sysconfig.get_config_var()`.

Configuration par l'environnement

Lors de l'initialisation de `TZPATH` (soit au moment de l'importation, soit lorsque `reset_tzpath()` est appelé sans argument), le module `zoneinfo` utilise la variable d'environnement `PYTHONTZPATH`, si elle existe, pour définir le chemin de recherche.

PYTHONTZPATH

Il s'agit d'une chaîne séparée par `os.pathsep` contenant le chemin de recherche du fuseau horaire à utiliser. Elle doit être constituée uniquement de chemins absolus et non de chemins relatifs. Les composants relatifs

spécifiés dans `PYTHONTZPATH` ne sont pas utilisés, mais le comportement reste défini par l'implémentation lorsqu'un chemin relatif est spécifié. CPython lève `InvalidTZPathWarning` dans ce cas. Les autres implémentations sont libres d'ignorer silencieusement le composant erroné ou de lever une exception.

Pour que le système ignore les données système et utilise le paquet *tzdata* à la place, définissez `PYTHONTZPATH=""`.

Configuration à l'exécution

Le chemin de recherche *TZ* peut également être configuré au moment de l'exécution à l'aide de la fonction `reset_tzpath()`. Cette opération n'est généralement pas conseillée, bien qu'il soit raisonnable de l'utiliser dans les fonctions de test qui nécessitent l'utilisation d'un chemin de fuseau horaire spécifique (ou qui nécessitent de désactiver l'accès aux fuseaux horaires du système).

8.2.3 La classe `ZoneInfo`

class zoneinfo.**ZoneInfo**(*key*)

Une sous-classe concrète de `datetime.tzinfo` qui représente un fuseau horaire IANA spécifié par la chaîne `key`. Les appels au constructeur principal renvoient toujours des objets dont la comparaison est identique ; autrement dit, sauf invalidation du cache via `ZoneInfo.clear_cache()`, pour toutes les valeurs de `key`, l'assertion suivante est toujours vraie :

```
a = ZoneInfo(key)
b = ZoneInfo(key)
assert a is b
```

`key` doit être sous la forme d'un chemin POSIX relatif et normalisé, sans références de niveau supérieur. Le constructeur lève `ValueError` si une clé non conforme est passée.

Si aucun fichier correspondant à `key` n'est trouvé, le constructeur lève `ZoneInfoNotFoundError`.

La classe `ZoneInfo` possède deux constructeurs alternatifs :

classmethod ZoneInfo.**from_file**(*fobj*, /, *key=None*)

Construit un objet `ZoneInfo` à partir d'un objet fichier renvoyant des octets (par exemple, un fichier ouvert en mode binaire ou un objet `io.BytesIO`). Contrairement au constructeur principal, il construit toujours un nouvel objet.

Le paramètre `key` définit le nom de la zone pour les besoins de `__str__()` et `__repr__()`.

Les objets créés via ce constructeur ne peuvent pas être sérialisés (voir *pickling*).

classmethod ZoneInfo.**no_cache**(*key*)

Un constructeur alternatif qui contourne le cache du constructeur. Il est identique au constructeur principal, mais renvoie un nouvel objet à chaque appel. Il est surtout utile à des fins de test ou de démonstration, mais il peut également être utilisé pour créer un système avec une stratégie d'invalidation du cache différente.

Les objets créés via ce constructeur contournent également le cache d'un processus de *désérialisation* lorsqu'ils sont désérialisés.

> **Prudence**
>
> L'utilisation de ce constructeur peut modifier la sémantique de vos *datetimes* de manière surprenante, ne l'utilisez que si vous en avez vraiment besoin.

Les méthodes de classe suivantes sont également disponibles :

classmethod ZoneInfo.**clear_cache**(*, *only_keys=None*)

Une méthode pour invalider le cache de la classe `ZoneInfo`. Si aucun argument n'est passé, tous les caches sont invalidés et le prochain appel au constructeur principal pour chaque clé renverra une nouvelle instance.

Si un itérable de noms de clés est passé au paramètre `only_keys`, seules les clés spécifiées sont retirées du cache. Les clés passées à `only_keys` mais non trouvées dans le cache sont ignorées.

> **Avertissement**
>
> Invoking this function may change the semantics of datetimes using `ZoneInfo` in surprising ways ; this modifies module state and thus may have wide-ranging effects. Only use it if you know that you need to.

La classe a un attribut :

`ZoneInfo.key`

Il s'agit d'un *attribut* en lecture seule qui renvoie la valeur de `key` passée au constructeur, qui doit être une clé de recherche dans la base de données des fuseaux horaires de l'IANA (par exemple, `America/New_York`, `Europe/Paris` ou `Asia/Tokyo`).

Pour les zones construites à partir d'un fichier sans spécifier de paramètre `key`, cette valeur sera fixée à `None`.

> **Note**
>
> Bien qu'il soit assez courant de les exposer aux utilisateurs finaux, ces valeurs sont conçues pour être des clés primaires permettant de représenter les zones concernées et pas nécessairement des éléments destinés à l'utilisateur. Des projets comme *CLDR* (*Unicode Common Locale Data Repository*) peuvent être utilisés pour obtenir des chaînes de caractères plus conviviales à partir de ces clés.

Représentation sous forme de chaîne de caractères

La représentation sous forme de chaîne renvoyée lors de l'appel de *`str`* sur un objet *`ZoneInfo`* utilise par défaut l'attribut *`ZoneInfo.key`* (voir la note sur l'utilisation dans la documentation de l'attribut) :

```
>>> zone = ZoneInfo("Pacific/Kwajalein")
>>> str(zone)
'Pacific/Kwajalein'

>>> dt = datetime(2020, 4, 1, 3, 15, tzinfo=zone)
>>> f"{dt.isoformat()} [{dt.tzinfo}]"
'2020-04-01T03:15:00+12:00 [Pacific/Kwajalein]'
```

Pour les objets construits à partir d'un fichier sans spécifier de paramètre `key`, `str` revient à appeler *`repr()`*. Le `repr` de `ZoneInfo` est défini par l'implémentation et n'est pas nécessairement stable entre les versions, mais il est garanti qu'il ne fournit pas un paramètre *key* `ZoneInfo` valide.

Sérialisation Pickle

Plutôt que de sérialiser toutes les données de transition, les objets `ZoneInfo` sont sérialisés à partir de l'attribut *key* et les objets `ZoneInfo` construits à partir de fichiers (même ceux avec une valeur pour `key` spécifiée) ne peuvent pas être sérialisés.

Le comportement d'un fichier `ZoneInfo` dépend de la façon dont il a été construit :

1. `ZoneInfo(key)` : lorsqu'il est construit avec le constructeur primaire, un objet `ZoneInfo` est sérialisé en utilisant l'attribut *key* et, lorsqu'il est désérialisé, le processus de désérialisation utilise le constructeur primaire et donc on s'attend à ce qu'il soit le même objet que d'autres références au même fuseau horaire. Par exemple, si `europe_berlin_pkl` est une chaîne contenant un objet sérialisé construit à partir de `ZoneInfo("Europe/Berlin")`, on s'attend au comportement suivant :

   ```
   >>> a = ZoneInfo("Europe/Berlin")
   >>> b = pickle.loads(europe_berlin_pkl)
   >>> a is b
   True
   ```

2. `ZoneInfo.no_cache(key)` : lorsqu'il est construit à partir du constructeur de contournement de cache, l'objet `ZoneInfo` est également sérialisé en utilisant l'attribut *key* mais, lorsqu'il est désérialisé, le processus

de désérialisation utilise le constructeur de contournement de cache. Si `europe_berlin_pkl_nc` est une chaîne contenant un objet sérialisé construit à partir de `ZoneInfo.no_cache("Europe/Berlin")`, le comportement suivant est attendu :

```
>>> a = ZoneInfo("Europe/Berlin")
>>> b = pickle.loads(europe_berlin_pkl_nc)
>>> a is b
False
```

3. `ZoneInfo.from_file(fobj, /, key=None)` : lorsqu'il est construit à partir d'un fichier, l'objet `ZoneInfo` lève une exception lors de la sérialisation. Si un utilisateur souhaite récupérer une `ZoneInfo` construite à partir d'un fichier, il est recommandé d'encapsuler les données dans un autre type ou d'utiliser une fonction de sérialisation personnalisée : soit en sérialisant en utilisant l'attribut *key*, soit en stockant le contenu de l'objet fichier et en sérialisant cet objet.

Cette méthode de sérialisation nécessite que les données de fuseau horaire relatives à l'attribut *key* soient disponibles à la fois du côté de la sérialisation et de la désérialisation, de la même manière que les références aux classes et aux fonctions sont censées exister dans les environnements de sérialisation et de désérialisation. Cela signifie également qu'aucune garantie n'est donnée quant à la cohérence des résultats lors de la désérialisation d'un `ZoneInfo` sérialisé dans un environnement avec une version différente des données de fuseau horaire.

8.2.4 Fonctions

`zoneinfo.`**`available_timezones`**`()`

Fournit un ensemble contenant toutes les clés valides pour les fuseaux horaires IANA disponibles n'importe où sur *TZPATH*. Il est recalculé à chaque appel à la fonction.

Cette fonction n'inclut que les noms de zone canoniques et n'inclut pas les zones « spéciales » telles que celles sous les répertoires `posix/` et `right/`, ou la zone `posixrules`.

> **Prudence**
>
> cette fonction peut ouvrir un grand nombre de fichiers, car la meilleure façon de déterminer si un fichier sur le chemin du fuseau horaire est un fuseau horaire valide est de lire la « chaîne magique » au début.

> **Note**
>
> ces valeurs n'ont pas vocation à être montrées aux utilisateurs ; pour les éléments destinés à l'utilisateur, les applications doivent utiliser quelque chose comme *CLDR* (le référentiel de données *Unicode Common Locale*) afin d'obtenir des chaînes plus conviviales. Voir aussi la note d'avertissement sur `ZoneInfo.key`.

`zoneinfo.`**`reset_tzpath`**`(`*to=None*`)`

Définit ou réinitialise le chemin de recherche des fuseaux horaires (`TZPATH`) pour le module. Lorsqu'il est appelé sans arguments, `TZPATH` est défini sur la valeur par défaut.

Appeler `reset_tzpath` n'invalide pas le cache `ZoneInfo`, donc les appels au constructeur primaire `ZoneInfo` n'utiliseront le nouveau `TZPATH` qu'en cas d'échec du cache.

Le paramètre `to` doit être une *séquence* de chaînes ou `os.PathLike` et non une chaîne, qui doivent toutes être des chemins absolus. `ValueError` est levée si autre chose qu'un chemin absolu est passé.

8.2.5 Variables globales

`zoneinfo.`**`TZPATH`**

Séquence en lecture seule représentant le chemin de recherche des fuseaux horaires – lors de la construction d'un `ZoneInfo` à partir de l'attribut *key*, la clé est jointe à chaque entrée dans le `TZPATH` et le premier fichier trouvé est utilisé.

`TZPATH` ne peut contenir que des chemins absolus, jamais des chemins relatifs, quelle que soit sa configuration.

L'objet vers lequel `zoneinfo.TZPATH` pointe peut changer suite à un appel à `reset_tzpath()`, il est donc recommandé d'utiliser `zoneinfo.TZPATH` plutôt que d'importer `TZPATH` de `zoneinfo` ou en assignant une variable de longue durée à `zoneinfo.TZPATH`.

Pour plus d'informations sur la configuration du chemin de recherche des fuseaux horaires, consultez *Configurer les sources de données*.

8.2.6 Exceptions et avertissements

exception zoneinfo.**ZoneInfoNotFoundError**

Levée lorsque la construction d'un objet `ZoneInfo` échoue parce que la clé spécifiée est introuvable sur le système. C'est une sous-classe de `KeyError`.

exception zoneinfo.**InvalidTZPathWarning**

Levée lorsque `PYTHONTZPATH` contient un composant non valide qui ne sera pas retenu, tel qu'un chemin relatif.

8.3 `calendar` --- General calendar-related functions

Code source : Lib/calendar.py

Ce module permet d'afficher un calendrier comme le fait le programme Unix **cal**, et il fournit des fonctions utiles relatives au calendrier. Par défaut, ces calendriers ont le lundi comme premier jour de la semaine et le dimanche comme dernier jour. Utilisez `setfirstweekday()` pour définir le premier jour de la semaine à dimanche (6) ou à tout autre jour de la semaine. Les paramètres pour spécifier les dates sont donnés sous forme de nombres entiers. Voir aussi les modules `datetime` et `time`.

Les fonctions et les classes définies dans ce module utilisent un calendrier idéalisé, le calendrier grégorien actuel s'étendant indéfiniment dans les deux sens. Cela correspond à la définition du calendrier grégorien proleptique dans le livre de Dershowitz et Reingold « *Calendrical Calculations* », œuvre dans lequel il est le calendrier de référence de tous les calculs. Les années zéros et les années négatives sont interprétées comme prescrit par la norme ISO 8601. L'année 0 est 1 avant JC, l'année `-1` est 2 avant JC et ainsi de suite.

class calendar.**Calendar**(*firstweekday=0*)

Crée un objet `Calendar`. *firstweekday* est un entier spécifiant le premier jour de la semaine, valant par défaut `0` (soit `MONDAY` pour lundi), pouvant aller jusqu'à `6` (soit `SUNDAY` pour dimanche).

L'objet `Calendar` fournit plusieurs méthodes pouvant être utilisées pour préparer les données du calendrier pour le formatage. Cette classe ne fait pas de formatage elle-même. Il s'agit du travail des sous-classes.

`Calendar` instances have the following methods and attributes :

firstweekday

The first weekday as an integer (0--6).

This property can also be set and read using `setfirstweekday()` and `getfirstweekday()` respectively.

getfirstweekday()

Return an `int` for the current first weekday (0--6).

Identical to reading the `firstweekday` property.

setfirstweekday(*firstweekday*)

Set the first weekday to *firstweekday*, passed as an `int` (0--6)

Identical to setting the `firstweekday` property.

iterweekdays()

Return an iterator for the week day numbers that will be used for one week. The first value from the iterator will be the same as the value of the `firstweekday` property.

itermonthdates(*year*, *month*)

Renvoie un itérateur sur les jours du mois *month* (1 à 12) de l'année *year*. Cet itérateur renvoie tous les jours du mois (sous forme d'instances de `datetime.date`) ainsi que tous les jours avant le début du mois et après la fin du mois nécessaires pour obtenir des semaines complètes.

itermonthdays(*year*, *month*)

Renvoie un itérateur sur les jours du mois *month* de l'année *year*, comme `itermonthdates()`, sans être limité par l'intervalle de `datetime.date`. Les jours renvoyés sont simplement les numéros des jours du mois. Pour les jours en dehors du mois spécifié, le numéro du jour est `0`.

itermonthdays2(*year*, *month*)

Renvoie un itérateur sur les jours du mois *month* de l'année *year* comme `itermonthdates()`, sans être limité par la plage `datetime.date`. Les jours renvoyés sont des paires composées du numéro du jour dans le mois et du numéro du jour dans la semaine.

itermonthdays3(*year*, *month*)

Renvoie un itérateur sur les jours du *month* de l'année *year*, comme `itermonthdates()`, sans être limité par l'intervalle de `datetime.date`. Les jours sont renvoyés sous forme de triplets composés du numéro de l'année, du mois et du jour dans le mois.

Ajouté dans la version 3.7.

itermonthdays4(*year*, *month*)

Renvoie un itérateur sur les jours du mois *month* de l'année *year*, comme `itermonthdates()`, sans être limité par l'intervalle de `datetime.date`. Le jours sont renvoyés sous forme de quadruplets contenant le numéro de l'année, du mois, du jour du mois et du jour de la semaine.

Ajouté dans la version 3.7.

monthdatescalendar(*year*, *month*)

Renvoie la liste des semaines complètes du mois *month* de l'année *year*. Les semaines sont des listes de sept objets `datetime.date`.

monthdays2calendar(*year*, *month*)

Renvoie la liste des semaines complètes du mois *month* de l'année *year*. Les semaines sont des listes de sept paires contenant le numéro du jour dans le mois et du numéro du jour dans la semaine.

monthdayscalendar(*year*, *month*)

Renvoie la liste des semaines complètes du mois *month* de l'année *year*. Les semaines sont une liste de sept numéros de jours.

yeardatescalendar(*year*, *width=3*)

Renvoie ce qu'il faut pour afficher correctement une année. La valeur renvoyée est une liste de lignes de mois. Chaque ligne mensuelle contient jusqu'à *width* mois (avec une valeur par défaut à 3). Chaque mois contient de 4 à 6 semaines et chaque semaine 1 à 7 jours. Les jours sont des objets `datetime.date`.

yeardays2calendar(*year*, *width=3*)

Renvoie ce qu'il faut pour afficher correctement une année, (similaire à `yeardatescalendar()`). Les listes des semaines contiennent des paires contenant le numéro du jour du mois et le numéro du jour de la semaine. Les numéro des jours en dehors de ce mois sont à zéro.

yeardayscalendar(*year*, *width=3*)

Renvoie ce qu'il faut pour afficher correctement une année, (similaire à `yeardatescalendar()`). Les listes de semaines contiennent des numéros de jours. Les numéros de jours en dehors de ce mois sont de zéro.

class calendar.**TextCalendar**(*firstweekday=0*)

Cette classe peut être utilisée pour générer des calendriers en texte brut.

Les instances `TextCalendar` exposent les méthodes suivantes :

formatday(*theday*, *weekday*, *width*)

Return a string representing a single day formatted with the given *width*. If *theday* is `0`, return a string of spaces of the specified width, representing an empty day. The *weekday* parameter is unused.

formatweek(*theweek*, *w=0*)

Return a single week in a string with no newline. If *w* is provided, it specifies the width of the date columns, which are centered. Depends on the first weekday as specified in the constructor or set by the `setfirstweekday()` method.

formatweekday(*weekday*, *width*)

Return a string representing the name of a single weekday formatted to the specified *width*. The *weekday* parameter is an integer representing the day of the week, where `0` is Monday and `6` is Sunday.

formatweekheader(*width*)

Return a string containing the header row of weekday names, formatted with the given *width* for each column. The names depend on the locale settings and are padded to the specified width.

formatmonth(*theyear*, *themonth*, *w=0*, *l=0*)

Donne le calendrier d'un mois dans une chaîne multi-ligne. Si *w* est fourni, il spécifie la largeur des colonnes de date, qui sont centrées. Si *l* est donné, il spécifie le nombre de lignes que chaque semaine utilisera. Le résultat varie en fonction du premier jour de la semaine spécifié dans le constructeur ou défini par la méthode `setfirstweekday()`.

formatmonthname(*theyear*, *themonth*, *width=0*, *withyear=True*)

Return a string representing the month's name centered within the specified *width*. If *withyear* is `True`, include the year in the output. The *theyear* and *themonth* parameters specify the year and month for the name to be formatted respectively.

prmonth(*theyear*, *themonth*, *w=0*, *l=0*)

Affiche le calendrier d'un mois tel que renvoyé par `formatmonth()`.

formatyear(*theyear*, *w=2*, *l=1*, *c=6*, *m=3*)

Renvoie un calendrier de *m* colonnes pour une année entière sous forme de chaîne multi-ligne. Les paramètres facultatifs *w*, *l* et *c* correspondent respectivement à la largeur de la colonne date, les lignes par semaines, le nombre d'espace entre les colonnes de mois. Le résultat varie en fonction du premier jour de la semaine spécifié dans le constructeur ou défini par la méthode `setfirstweekday()`. La première année pour laquelle un calendrier peut être généré, dépend de la plateforme.

pryear(*theyear*, *w=2*, *l=1*, *c=6*, *m=3*)

Affiche le calendrier pour une année entière comme renvoyé par `formatyear()`.

class calendar.**HTMLCalendar**(*firstweekday=0*)

Cette classe peut être utilisée pour générer des calendriers HTML.

Les instances de `HTMLCalendar` utilisent les méthodes suivantes :

formatmonth(*theyear*, *themonth*, *withyear=True*)

Renvoie le calendrier d'un mois sous la forme d'une table HTML. Si *withyear* est vrai l'année sera inclue dans l'en-tête, sinon seul le nom du mois sera utilisé.

formatyear(*theyear*, *width=3*)

Renvoie le calendrier d'une année sous la forme d'une table HTML. *width* (par défaut à 3) spécifie le nombre de mois par ligne.

formatyearpage(*theyear*, *width=3*, *css='calendar.css'*, *encoding=None*)

Renvoie le calendrier d'une année sous la forme d'une page HTML complète. *width* (par défaut à 3) spécifie le nombre de mois par ligne. *css* est le nom de la feuille de style en cascade à utiliser. `None` peut être passé si aucune feuille de style ne doit être utilisée. *encoding* spécifie l'encodage à utiliser pour les données de sortie (par défaut l'encodage par défaut du système).

formatmonthname(*theyear*, *themonth*, *withyear=True*)

Return a month name as an HTML table row. If *withyear* is true the year will be included in the row, otherwise just the month name will be used.

`HTMLCalendar` possède les attributs suivants que vous pouvez surcharger pour personnaliser les classes CSS utilisées par le calendrier :

cssclasses

Une liste de classes CSS utilisées pour chaque jour de la semaine. La liste par défaut de la classe est :

```
cssclasses = ["mon", "tue", "wed", "thu", "fri", "sat", "sun"]
```

davantage de styles peuvent être ajoutés pour chaque jour :

```
cssclasses = ["mon text-bold", "tue", "wed", "thu", "fri", "sat", "sun red
↪"]
```

Notez que la longueur de cette liste doit être de sept éléments.

cssclass_noday

La classe CSS pour le jour de la semaine apparaissant dans le mois précédent ou à venir.

Ajouté dans la version 3.7.

cssclasses_weekday_head

Une liste de classes CSS utilisées pour les noms des jours de la semaine dans la ligne d'en-tête. Par défaut les mêmes que *cssclasses*.

Ajouté dans la version 3.7.

cssclass_month_head

La classe CSS du mois en en-tête (utilisé par *formatmonthname()*). La valeur par défaut est `"month"`.

Ajouté dans la version 3.7.

cssclass_month

La classe CSS pour la table du mois entière (utilisé par *formatmonth()*). La valeur par défaut est `"month"`.

Ajouté dans la version 3.7.

cssclass_year

La classe CSS pour la table entière des tables de l'année (utilisé par *formatyear()*). La valeur par défaut est `"year"`.

Ajouté dans la version 3.7.

cssclass_year_head

La classe CSS pour l'en-tête de la table pour l'année entière (utilisé par *formatyear()*). La valeur par défaut est `"year"`.

Ajouté dans la version 3.7.

Notez que même si le nommage ci-dessus des attributs de classe est au singulier (p. ex. `cssclass_month` `cssclass_noday`), on peut remplacer la seule classe CSS par une liste de classes CSS séparées par une espace, par exemple :

```
"text-bold text-red"
```

Voici un exemple de comment peut être personnalisée `HTMLCalendar` :

```
class CustomHTMLCal(calendar.HTMLCalendar):
    cssclasses = [style + " text-nowrap" for style in
                  calendar.HTMLCalendar.cssclasses]
```

(suite sur la page suivante)

(suite de la page précédente)

```
    cssclass_month_head = "text-center month-head"
    cssclass_month = "text-center month"
    cssclass_year = "text-italic lead"
```

class calendar.**LocaleTextCalendar**(*firstweekday=0*, *locale=None*)

Il est possible de passer un nom de locale au constructeur de la sous-classe de `TextCalendar`. Elle renvoie les noms des mois et des jours de la semaine dans la locale précisée.

class calendar.**LocaleHTMLCalendar**(*firstweekday=0*, *locale=None*)

Il est possible de passer un nom de locale au constructeur de la sous-classe de `HTMLCalendar`. Elle renvoie les noms des mois et des jours de la semaine dans la locale précisée.

Note

The constructor, `formatweekday()` and `formatmonthname()` methods of these two classes temporarily change the `LC_TIME` locale to the given *locale*. Because the current locale is a process-wide setting, they are not thread-safe.

Pour les calendriers texte simples ce module fournit les fonctions suivantes.

calendar.**setfirstweekday**(*weekday*)

Fixe le jour de la semaine (`0` pour lundi, `6` pour dimanche) qui débute chaque semaine. Les valeurs `MONDAY`, `TUESDAY`, `WEDNESDAY`, `THURSDAY`, `FRIDAY`, `SATURDAY`, et `SUNDAY` sont fournies par commodité. Par exemple, pour fixer le premier jour de la semaine à dimanche :

```
import calendar
calendar.setfirstweekday(calendar.SUNDAY)
```

calendar.**firstweekday**()

Renvoie le réglage courant pour le jour de la semaine débutant chaque semaine.

calendar.**isleap**(*year*)

Renvoie `True` si *year* est une année bissextile, sinon `False`.

calendar.**leapdays**(*y1*, *y2*)

Renvoie le nombre d'années bissextiles dans la période de *y1* à *y2* (non inclus), où *y1* et *y2* sont des années.

Cette fonction marche pour les périodes couvrant un changement de siècle.

calendar.**weekday**(*year*, *month*, *day*)

Renvoie le jour de la semaine (`0` pour lundi) pour *year* (`1970`-- …), *month* (`1`--`12`), *day* (`1`--`31`).

calendar.**weekheader**(*n*)

Renvoie un en-tête contenant les jours de la semaine en abrégé. *n* spécifie la largeur en caractères pour un jour de la semaine.

calendar.**monthrange**(*year*, *month*)

Renvoie le jour de la semaine correspondant au premier jour du mois et le nombre de jours dans le mois, pour l'année *year* et le mois *month* spécifiés.

calendar.**monthcalendar**(*year*, *month*)

Renvoie une matrice représentant le calendrier d'un mois. Chaque ligne représente une semaine ; les jours en dehors du mois sont représentés par des zéros. Chaque semaine débute avec le lundi à moins de l'avoir modifié avec `setfirstweekday()`.

calendar.**prmonth**(*theyear*, *themonth*, *w=0*, *l=0*)

Affiche le calendrier d'un mois tel que renvoyé par `month()`.

calendar.**month**(*theyear*, *themonth*, *w=0*, *l=0*)

Returns a month's calendar in a multi-line string using the `formatmonth()` of the `TextCalendar` class.

calendar.**prcal**(*year*, *w=0*, *l=0*, *c=6*, *m=3*)

Affiche le calendrier pour une année entière tel que renvoyé par `calendar()`.

calendar.**calendar**(*year*, *w=2*, *l=1*, *c=6*, *m=3*)

Returns a 3-column calendar for an entire year as a multi-line string using the `formatyear()` of the `TextCalendar` class.

calendar.**timegm**(*tuple*)

Une fonction sans rapport mais pratique, qui prend un *n*-uplet temporel tel que celui renvoyé par la fonction `gmtime()` dans le module `time`, et renvoie la valeur d'horodatage Unix (*timestamp* en anglais) correspondante, en supposant une époque de 1970, et l'encodage POSIX. En fait, `time.gmtime()` et `timegm()` sont l'inverse l'un de l'autre.

Le module `calendar` exporte les attributs suivants :

calendar.**day_name**

A sequence that represents the days of the week in the current locale, where Monday is day number 0.

```
>>> import calendar
>>> list(calendar.day_name)
['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday']
```

calendar.**day_abbr**

A sequence that represents the abbreviated days of the week in the current locale, where Mon is day number 0.

```
>>> import calendar
>>> list(calendar.day_abbr)
['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']
```

calendar.**MONDAY**

calendar.**TUESDAY**

calendar.**WEDNESDAY**

calendar.**THURSDAY**

calendar.**FRIDAY**

calendar.**SATURDAY**

calendar.**SUNDAY**

Aliases for the days of the week, where `MONDAY` is `0` and `SUNDAY` is `6`.

Ajouté dans la version 3.12.

class calendar.**Day**

Enumeration defining days of the week as integer constants. The members of this enumeration are exported to the module scope as `MONDAY` through `SUNDAY`.

Ajouté dans la version 3.12.

calendar.**month_name**

A sequence that represents the months of the year in the current locale. This follows normal convention of January being month number 1, so it has a length of 13 and `month_name[0]` is the empty string.

```
>>> import calendar
>>> list(calendar.month_name)
['', 'January', 'February', 'March', 'April', 'May', 'June', 'July', 'August',
↪'September', 'October', 'November', 'December']
```

calendar.**month_abbr**

A sequence that represents the abbreviated months of the year in the current locale. This follows normal convention of January being month number 1, so it has a length of 13 and `month_abbr[0]` is the empty string.

```
>>> import calendar
>>> list(calendar.month_abbr)
['', 'Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov
↪', 'Dec']
```

calendar.**JANUARY**

calendar.**FEBRUARY**

calendar.**MARCH**

calendar.**APRIL**

calendar.**MAY**

calendar.**JUNE**

calendar.**JULY**

calendar.**AUGUST**

calendar.**SEPTEMBER**

calendar.**OCTOBER**

calendar.**NOVEMBER**

calendar.**DECEMBER**

Aliases for the months of the year, where `JANUARY` is `1` and `DECEMBER` is `12`.

Ajouté dans la version 3.12.

class calendar.**Month**

Enumeration defining months of the year as integer constants. The members of this enumeration are exported to the module scope as *JANUARY* through *DECEMBER*.

Ajouté dans la version 3.12.

The *calendar* module defines the following exceptions :

exception calendar.**IllegalMonthError**(*month*)

A subclass of *ValueError*, raised when the given month number is outside of the range 1-12 (inclusive).

month

The invalid month number.

exception calendar.**IllegalWeekdayError**(*weekday*)

A subclass of *ValueError*, raised when the given weekday number is outside of the range 0-6 (inclusive).

weekday

The invalid weekday number.

Voir aussi

Module *datetime*
Interface orientée objet pour les dates et les heures avec des fonctionnalités similaires au module *time*.

Module *time*
Fonctions bas niveau relatives au temps.

8.3.1 Command-line usage

Ajouté dans la version 2.5.

The `calendar` module can be executed as a script from the command line to interactively print a calendar.

```
python -m calendar [-h] [-L LOCALE] [-e ENCODING] [-t {text,html}]
                   [-w WIDTH] [-l LINES] [-s SPACING] [-m MONTHS] [-c CSS]
                   [-f FIRST_WEEKDAY] [year] [month]
```

For example, to print a calendar for the year 2000 :

```
$ python -m calendar 2000
                                  2000

      January                   February                   March
Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su
                1  2          1  2  3  4  5  6             1  2  3  4  5
 3  4  5  6  7  8  9       7  8  9 10 11 12 13       6  7  8  9 10 11 12
10 11 12 13 14 15 16      14 15 16 17 18 19 20      13 14 15 16 17 18 19
17 18 19 20 21 22 23      21 22 23 24 25 26 27      20 21 22 23 24 25 26
24 25 26 27 28 29 30      28 29                     27 28 29 30 31
31

       April                     May                       June
Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su
                1  2       1  2  3  4  5  6  7                1  2  3  4
 3  4  5  6  7  8  9       8  9 10 11 12 13 14       5  6  7  8  9 10 11
10 11 12 13 14 15 16      15 16 17 18 19 20 21      12 13 14 15 16 17 18
17 18 19 20 21 22 23      22 23 24 25 26 27 28      19 20 21 22 23 24 25
24 25 26 27 28 29 30      29 30 31                  26 27 28 29 30

        July                    August                  September
Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su
                1  2          1  2  3  4  5  6                   1  2  3
 3  4  5  6  7  8  9       7  8  9 10 11 12 13       4  5  6  7  8  9 10
10 11 12 13 14 15 16      14 15 16 17 18 19 20      11 12 13 14 15 16 17
17 18 19 20 21 22 23      21 22 23 24 25 26 27      18 19 20 21 22 23 24
24 25 26 27 28 29 30      28 29 30 31               25 26 27 28 29 30
31

      October                   November                  December
Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su
                   1             1  2  3  4  5                   1  2  3
 2  3  4  5  6  7  8       6  7  8  9 10 11 12       4  5  6  7  8  9 10
 9 10 11 12 13 14 15      13 14 15 16 17 18 19      11 12 13 14 15 16 17
16 17 18 19 20 21 22      20 21 22 23 24 25 26      18 19 20 21 22 23 24
23 24 25 26 27 28 29      27 28 29 30               25 26 27 28 29 30 31
30 31
```

Les options suivantes sont acceptées :

--help, -h

Affiche un message d'aide et quitte.

--locale LOCALE, **-L** LOCALE

The locale to use for month and weekday names. Defaults to English.

--encoding ENCODING, **-e** ENCODING

The encoding to use for output. `--encoding` is required if `--locale` is set.

--type {text,html}, **-t** {text,html}

Print the calendar to the terminal as text, or as an HTML document.

--first-weekday FIRST_WEEKDAY, **-f** FIRST_WEEKDAY

The weekday to start each week. Must be a number between 0 (Monday) and 6 (Sunday). Defaults to 0.

Ajouté dans la version 3.13.

year

The year to print the calendar for. Defaults to the current year.

month

The month of the specified *year* to print the calendar for. Must be a number between 1 and 12, and may only be used in text mode. Defaults to printing a calendar for the full year.

Text-mode options :

--width WIDTH, **-w** WIDTH

The width of the date column in terminal columns. The date is printed centred in the column. Any value lower than 2 is ignored. Defaults to 2.

--lines LINES, **-l** LINES

The number of lines for each week in terminal rows. The date is printed top-aligned. Any value lower than 1 is ignored. Defaults to 1.

--spacing SPACING, **-s** SPACING

The space between months in columns. Any value lower than 2 is ignored. Defaults to 6.

--months MONTHS, **-m** MONTHS

The number of months printed per row. Defaults to 3.

Modifié dans la version 3.14 : By default, today's date is highlighted in color and can be controlled using environment variables.

HTML-mode options :

--css CSS, **-c** CSS

The path of a CSS stylesheet to use for the calendar. This must either be relative to the generated HTML, or an absolute HTTP or `file:///` URL.

8.4 `collections` --- Container datatypes

Code source : Lib/collections/__init__.py

Ce module implémente des types de données de conteneurs spécialisés qui apportent des alternatives aux conteneurs natifs de Python plus généraux *dict*, *list*, *set* et *tuple*.

namedtuple()	fonction permettant de créer des sous-classes de `tuple` avec des champs nommés
deque	conteneur se comportant comme une liste avec des ajouts et retraits rapides à chaque extrémité
ChainMap	classe semblable aux dictionnaires qui crée une unique vue à partir de plusieurs dictionnaires
Counter	dict subclass for counting *hashable* objects
OrderedDict	sous-classe de `dict` qui garde en mémoire l'ordre dans lequel les entrées ont été ajoutées
defaultdict	sous-classe de `dict` qui appelle une fonction de fabrication en cas de valeur manquante
UserDict	surcouche autour des objets dictionnaires pour faciliter l'héritage de `dict`
UserList	surcouche autour des objets listes pour faciliter l'héritage de `list`
UserString	surcouche autour des objets chaînes de caractères pour faciliter l'héritage de `str`

8.4.1 Objets `ChainMap`

Ajouté dans la version 3.3.

Le module fournit une classe `ChainMap` afin de réunir rapidement plusieurs dictionnaires en une unique entité. Cela est souvent plus rapide que de créer un nouveau dictionnaire et d'effectuer plusieurs appels de `update()`.

Cette classe peut être utilisée pour simuler des portées imbriquées, elle est aussi utile pour le *templating*.

class `collections.`**`ChainMap`**(**maps*)

Un objet `ChainMap` regroupe plusieurs dictionnaires (ou autres tableaux de correspondance) en une vue que l'on peut mettre à jour. Si le paramètre *maps* est vide, un dictionnaire vide est fourni de telle manière qu'une nouvelle chaîne possède toujours au moins un dictionnaire.

Les dictionnaires sous-jacents sont stockés dans une liste. Celle-ci est publique et peut être consultée ou mise à jour via l'attribut *maps*. Il n'y a pas d'autre état.

Les recherches s'effectuent successivement dans chaque dictionnaire jusqu'à la première clé correspondante. En revanche, les écritures, mises à jour et suppressions n'affectent que le premier dictionnaire.

Un objet `ChainMap` incorpore les dictionnaires sous-jacents par leur référence. Ainsi, si l'un d'eux est modifié, les changements affectent également la `ChainMap`.

Toutes les méthodes usuelles des dictionnaires sont gérées. De plus, cette classe fournit un attribut *maps*, une méthode pour créer de nouveaux sous-contextes et une propriété pour accéder à tous les dictionnaires sous-jacents excepté le premier :

`maps`

Liste de dictionnaires éditable par l'utilisateur et classée selon l'ordre de recherche. Il s'agit de l'unique état stocké et elle peut être modifiée pour changer l'ordre de recherche. La liste doit toujours contenir au moins un dictionnaire.

`new_child`(*m=None*, ***kwargs*)

Renvoie un nouvel objet `ChainMap` contenant un nouveau dictionnaire suivi par tous les autres de l'instance actuelle. Si `m` est spécifié, il devient le nouveau dictionnaire au début de la liste ; sinon, un dictionnaire vide est utilisé, de telle manière qu'appeler `d.new_child()` équivaut à appeler `ChainMap({}, *d.maps)`. Si des arguments sont passés par mot-clé, ils sont insérés comme de nouvelles entrées du dictionnaire ajouté. Cette méthode est utile pour créer des sous-contextes qui peuvent être mis à jour sans altérer les valeurs dans les dictionnaires parents.

Modifié dans la version 3.4 : ajout du paramètre optionnel `m`.

Modifié dans la version 3.10 : prise en charge des arguments par mot-clé.

`parents`

Propriété qui renvoie un nouvel objet `ChainMap` contenant tous les dictionnaires de l'instance actuelle hormis le premier. Cette propriété est utile pour ignorer le premier dictionnaire dans les recherches ; son utilisation rappelle le mot-clé `nonlocal` (utilisé pour les *portées imbriquées*), ou bien la fonction native `super()`. Une référence à `d.parents` est équivalente à : `ChainMap(*d.maps[1:])`.

Note, the iteration order of a `ChainMap` is determined by scanning the mappings last to first :

```
>>> baseline = {'music': 'bach', 'art': 'rembrandt'}
>>> adjustments = {'art': 'van gogh', 'opera': 'carmen'}
>>> list(ChainMap(adjustments, baseline))
['music', 'art', 'opera']
```

Cela produit le même ordre qu'une suite d'appels à `dict.update()` en commençant par le dernier tableau de correspondances :

```
>>> combined = baseline.copy()
>>> combined.update(adjustments)
>>> list(combined)
['music', 'art', 'opera']
```

Modifié dans la version 3.9 : Ajout de la gestion des opérateurs `|` et `|=` tels que définis dans **PEP 584**.

Voir aussi

- La classe MultiContext dans le package CodeTools d'Enthought possède des options pour gérer l'écriture dans n'importe quel dictionnaire de la chaîne.
- La classe de contexte de Django pour la création de modèles est une chaîne de dictionnaires en lecture seule. Elle comporte également des fonctionnalités d'ajouts et de retraits de contextes similaires à la méthode `new_child()` et à la propriété `parents`.
- The Nested Contexts recipe has options to control whether writes and other mutations apply only to the first mapping or to any mapping in the chain.
- Une version grandement simplifiée de Chainmap en lecture seule.

Exemples et cas pratiques utilisant `ChainMap`

Cette partie montre diverses approches afin de travailler avec les dictionnaires chaînés.

Exemple 1 : simulation de la chaîne de recherche interne de Python :

```
import builtins
pylookup = ChainMap(locals(), globals(), vars(builtins))
```

Exemple 2 : spécification d'une hiérarchie pour les options : ligne de commande, variable d'environnement, valeurs par défaut :

```
import os, argparse

defaults = {'color': 'red', 'user': 'guest'}

parser = argparse.ArgumentParser()
parser.add_argument('-u', '--user')
parser.add_argument('-c', '--color')
namespace = parser.parse_args()
command_line_args = {k: v for k, v in vars(namespace).items() if v is not None}

combined = ChainMap(command_line_args, os.environ, defaults)
print(combined['color'])
print(combined['user'])
```

Exemple 3 : modèles pour simuler des contextes imbriqués avec la classe `ChainMap` :

```
c = ChainMap()          # Create root context
d = c.new_child()       # Create nested child context
e = c.new_child()       # Child of c, independent from d
e.maps[0]               # Current context dictionary -- like Python's locals()
e.maps[-1]              # Root context -- like Python's globals()
e.parents               # Enclosing context chain -- like Python's nonlocals

d['x'] = 1              # Set value in current context
d['x']                  # Get first key in the chain of contexts
del d['x']              # Delete from current context
list(d)                 # All nested values
k in d                  # Check all nested values
len(d)                  # Number of nested values
d.items()               # All nested items
dict(d)                 # Flatten into a regular dictionary
```

La classe `ChainMap` ne met à jour (écriture et suppression) que le premier dictionnaire de la chaîne, alors qu'une

recherche inspecte toute la chaîne. Cependant, si l'on veut effectuer des écritures ou suppressions en profondeur, on peut facilement faire une sous-classe qui met à jour les clés trouvées de la chaîne en profondeur :

```
class DeepChainMap(ChainMap):
    'Variant of ChainMap that allows direct updates to inner scopes'

    def __setitem__(self, key, value):
        for mapping in self.maps:
            if key in mapping:
                mapping[key] = value
                return
        self.maps[0][key] = value

    def __delitem__(self, key):
        for mapping in self.maps:
            if key in mapping:
                del mapping[key]
                return
        raise KeyError(key)

>>> d = DeepChainMap({'zebra': 'black'}, {'elephant': 'blue'}, {'lion': 'yellow'})
>>> d['lion'] = 'orange'         # update an existing key two levels down
>>> d['snake'] = 'red'           # new keys get added to the topmost dict
>>> del d['elephant']            # remove an existing key one level down
>>> d                            # display result
DeepChainMap({'zebra': 'black', 'snake': 'red'}, {}, {'lion': 'orange'})
```

8.4.2 Objets `Counter`

Ce module fournit un outil pour effectuer rapidement et facilement des dénombrements. Par exemple :

```
>>> # Tally occurrences of words in a list
>>> cnt = Counter()
>>> for word in ['red', 'blue', 'red', 'green', 'blue', 'blue']:
...     cnt[word] += 1
...
>>> cnt
Counter({'blue': 3, 'red': 2, 'green': 1})

>>> # Find the ten most common words in Hamlet
>>> import re
>>> words = re.findall(r'\w+', open('hamlet.txt').read().lower())
>>> Counter(words).most_common(10)
[('the', 1143), ('and', 966), ('to', 762), ('of', 669), ('i', 631),
 ('you', 554),  ('a', 546), ('my', 514), ('hamlet', 471), ('in', 451)]
```

class collections.**Counter**([*iterable-or-mapping*])

A `Counter` is a `dict` subclass for counting *hashable* objects. It is a collection where elements are stored as dictionary keys and their counts are stored as dictionary values. Counts are allowed to be any integer value including zero or negative counts. The `Counter` class is similar to bags or multisets in other languages.

Les éléments sont comptés à partir d'un itérable ou initialisés à partir d'un autre dictionnaire (ou compteur) :

```
>>> c = Counter()                           # a new, empty counter
>>> c = Counter('gallahad')                 # a new counter from an iterable
>>> c = Counter({'red': 4, 'blue': 2})      # a new counter from a mapping
>>> c = Counter(cats=4, dogs=8)             # a new counter from keyword args
```

Les objets Counter ont une interface de dictionnaire, à l'exception près qu'ils renvoient zéro au lieu de lever une exception `KeyError` pour des éléments manquants :

```
>>> c = Counter(['eggs', 'ham'])
>>> c['bacon']                              # count of a missing element is␣
↪zero
0
```

Mettre un comptage à zéro pour un élément ne le retire pas de l'objet Counter. Il faut utiliser `del` pour le supprimer complètement :

```
>>> c['sausage'] = 0                        # counter entry with a zero count
>>> del c['sausage']                        # del actually removes the entry
```

Ajouté dans la version 3.1.

Modifié dans la version 3.7 : As a `dict` subclass, `Counter` inherited the capability to remember insertion order. Math operations on *Counter* objects also preserve order. Results are ordered according to when an element is first encountered in the left operand and then by the order encountered in the right operand.

Counter objects support additional methods beyond those available for all dictionaries :

elements()

Renvoie un itérateur sur chaque élément en le répétant autant de fois que la valeur du compteur associé. Les éléments sont renvoyés dans l'ordre selon lequel ils sont rencontrés pour la première fois. Si le comptage d'un élément est strictement inférieur à 1, alors `elements()` l'ignore.

```
>>> c = Counter(a=4, b=2, c=0, d=-2)
>>> sorted(c.elements())
['a', 'a', 'a', 'a', 'b', 'b']
```

most_common([*n*])

Renvoie une liste des *n* éléments les plus nombreux et leur valeur respective dans l'ordre décroissant. Si *n* n'est pas fourni ou vaut `None`, `most_common()` renvoie *tous* les éléments du compteur. Les éléments qui ont le même nombre d'occurrences sont ordonnés par l'ordre selon lequel ils ont été rencontrés pour la première fois :

```
>>> Counter('abracadabra').most_common(3)
[('a', 5), ('b', 2), ('r', 2)]
```

subtract([*iterable-or-mapping*])

Les éléments sont soustraits à partir d'un itérable ou d'un autre dictionnaire (ou compteur). Cette méthode se comporte comme `dict.update()` mais soustrait les nombres d'occurrences au lieu de les remplacer. Les entrées et sorties peuvent être négatives ou nulles.

```
>>> c = Counter(a=4, b=2, c=0, d=-2)
>>> d = Counter(a=1, b=2, c=3, d=4)
>>> c.subtract(d)
>>> c
Counter({'a': 3, 'b': 0, 'c': -3, 'd': -6})
```

Ajouté dans la version 3.2.

total()

Calcule la somme totale des nombres d'occurrences.

```
>>> c = Counter(a=10, b=5, c=0)
>>> c.total()
15
```

Ajouté dans la version 3.10.

Les méthodes usuelles des dictionnaires sont disponibles pour les objets `Counter` à l'exception de deux méthodes qui fonctionnent différemment pour les compteurs.

fromkeys(*iterable*)

Cette méthode de classe n'est pas implémentée pour les objets `Counter`.

update([*iterable-or-mapping*])

Les éléments sont comptés à partir d'un itérable ou ajoutés d'un autre dictionnaire (ou compteur). Cette méthode se comporte comme `dict.update()` mais additionne les nombres d'occurrences au lieu de les remplacer. De plus, l'itérable doit être une séquence d'éléments et non une séquence de paires `(clé, valeur)`.

Les compteurs prennent en charge les comparaisons riches pour les tests d'égalité ainsi que d'inclusion du membre gauche dans le membre droite et réciproquement, avec les opérateurs ==, !=, <, <=, > et >=. Les éléments dont le nombre d'occurrences est à zéro sont ignorés. Par exemple, on a `Counter(a=1) == Counter(a=1, b=0)`.

Modifié dans la version 3.10 : ajout des comparaisons riches.

Modifié dans la version 3.10 : les éléments dont le nombre d'occurrences est à zéro sont désormais ignorés dans les tests d'égalité. On avait auparavant `Counter(a=3) != Counter(a=3, b=0)`.

Opérations usuelles sur les objets `Counter` :

```
c.total()                       # total of all counts
c.clear()                       # reset all counts
list(c)                         # list unique elements
set(c)                          # convert to a set
dict(c)                         # convert to a regular dictionary
c.items()                       # access the (elem, cnt) pairs
Counter(dict(list_of_pairs))    # convert from a list of (elem, cnt) pairs
c.most_common()[:-n-1:-1]       # n least common elements
+c                              # remove zero and negative counts
```

Several mathematical operations are provided for combining `Counter` objects to produce multisets (counters that have counts greater than zero). Addition and subtraction combine counters by adding or subtracting the counts of corresponding elements. Intersection and union return the minimum and maximum of corresponding counts. Equality and inclusion compare corresponding counts. Each operation can accept inputs with signed counts, but the output will exclude results with counts of zero or less.

```
>>> c = Counter(a=3, b=1)
>>> d = Counter(a=1, b=2)
>>> c + d                       # add two counters together:  c[x] + d[x]
Counter({'a': 4, 'b': 3})
>>> c - d                       # subtract (keeping only positive counts)
Counter({'a': 2})
>>> c & d                       # intersection:  min(c[x], d[x])
Counter({'a': 1, 'b': 1})
>>> c | d                       # union:  max(c[x], d[x])
Counter({'a': 3, 'b': 2})
>>> c == d                      # equality:  c[x] == d[x]
False
>>> c <= d                      # inclusion:  c[x] <= d[x]
False
```

L'addition et la soustraction unaires (avec un seul terme) sont des raccourcis pour respectivement additionner un compteur avec un compteur vide ou et pour retrancher un compteur d'un compteur vide.

```
>>> c = Counter(a=2, b=-4)
>>> +c
Counter({'a': 2})
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> -c
Counter({'b': 4})
```

Ajouté dans la version 3.3 : Ajout de la gestion des additions et soustractions unaires, et des remplacements dans les multiensembles.

Note

Les compteurs ont été conçus essentiellement pour fonctionner avec des entiers naturels pour représenter les dénombrements en cours ; cependant, les cas d'utilisation nécessitant d'autres types ou des valeurs négatives n'ont pas été écartés. Pour vous aider dans ces cas particuliers, cette section documente la plage minimale et les restrictions de type.

- La classe `Counter` est elle-même une sous-classe de dictionnaire sans restriction particulière sur ces clés ou valeurs. Les valeurs ont vocation à être des nombres représentants des comptages, mais il est *possible* de stocker n'importe quel type de valeur.
- La méthode `most_common()` exige uniquement que les valeurs soient ordonnables.
- Les opérations de remplacement telles que `c[key] += 1` exigent une valeur dont le type gère l'addition et la soustraction. Cela inclut donc les fractions, les flottants et les décimaux, y compris négatifs. Il en va de même pour `update()` et `substract()` qui acceptent des valeurs négatives ou nulles dans les entrées et sorties.
- Les méthodes de multiensembles sont uniquement conçues pour les cas d'utilisation avec des valeurs positives. Les entrées peuvent contenir des valeurs négatives ou nulles, mais seules les sorties avec des valeurs positives sont créées. Il n'y a pas de restriction de type, mais les types des valeurs doivent gérer l'addition, la soustraction et la comparaison.
- La méthode `elements()` exige des valeurs entières et ignore les valeurs négatives ou nulles.

Voir aussi

- Bag class dans Smalltalk.
- L'article Wikipédia sur les multiensembles sur Wikipédia (ou l'article en anglais).
- Des guides et exemples à propos des multiensembles en C++.
- Pour les opérations mathématiques sur les multiensembles et leurs applications, voir *Knuth, Donald. The Art of Computer Programming Volume II, Section 4.6.3, Exercise 19*.
- Pour lister tous les multiensembles distincts de même taille parmi un ensemble donné d'éléments, voir `itertools.combinations_with_replacement()` :

```
map(Counter, combinations_with_replacement('ABC', 2)) # --> AA AB AC BB BC␣
↪CC
```

8.4.3 Objets deque

class `collections.deque`([*iterable*[, *maxlen*]])

Renvoie un nouvel objet *deque* initialisé de gauche à droite (en utilisant `append()`) avec les données d'*iterable*. Si *iterable* n'est pas spécifié, alors la nouvelle *deque* est vide.

Deques are a generalization of stacks and queues (the name is pronounced "deck" and is short for "double-ended queue"). Deques support thread-safe, memory efficient appends and pops from either side of the deque with approximately the same $O(1)$ performance in either direction.

Though `list` objects support similar operations, they are optimized for fast fixed-length operations and incur $O(n)$ memory movement costs for `pop(0)` and `insert(0, v)` operations which change both the size and position of the underlying data representation.

Si *maxlen* n'est pas spécifié ou vaut *None*, les *deques* peuvent atteindre une taille arbitraire. Sinon, la *deque* est limitée par cette taille maximale. Une fois que celle-ci est atteinte, un ajout d'un ou plusieurs éléments engendre la suppression du nombre correspondant d'éléments à l'autre extrémité de la *deque*. Les *deques* à

longueur limitée apportent des fonctionnalités similaires au filtre `tail` d'Unix. Elles sont aussi utiles pour le suivi de transactions et autres lots de données où seule l'activité récente est intéressante.

Les objets *deques* gèrent les méthodes suivantes :

append(*x*)

Ajoute *x* à l'extrémité droite de la *deque*.

appendleft(*x*)

Ajoute *x* à l'extrémité gauche de la *deque*.

clear()

Supprime tous les éléments de la *deque* et la laisse avec une longueur de 0.

copy()

Crée une copie superficielle de la *deque*.

Ajouté dans la version 3.5.

count(*x*)

Compte le nombre d'éléments de la *deque* égaux à *x*.

Ajouté dans la version 3.2.

extend(*iterable*)

Étend la *deque* en ajoutant les éléments de l'itérable en argument à son extrémité droite.

extendleft(*iterable*)

Étend la *deque* en ajoutant les éléments d'*iterable* à son extrémité gauche. Dans ce cas, notez que la série d'ajouts inverse l'ordre des éléments de l'argument itérable.

index(*x*[, *start*[, *stop*]])

Renvoie la position de *x* dans la *deque* (à partir de *start* inclus et jusqu'à *stop* exclus). Renvoie la première correspondance ou lève `ValueError` si aucune n'est trouvée.

Ajouté dans la version 3.5.

insert(*i*, *x*)

Insère *x* dans la *deque* à la position *i*.

Si une insertion provoque un dépassement de la taille limitée d'une *deque*, alors elle lève une exception `IndexError`.

Ajouté dans la version 3.5.

pop()

Retire et renvoie un élément de l'extrémité droite de la *deque*. S'il n'y a aucun élément, lève une exception `IndexError`.

popleft()

Retire et renvoie un élément de l'extrémité gauche de la *deque*. S'il n'y a aucun élément, lève une exception `IndexError`.

remove(*value*)

Supprime la première occurrence de *value*. Si aucune occurrence n'est trouvée, lève une exception `ValueError`.

reverse()

Inverse le sens des éléments de la *deque* sans créer de copie et renvoie `None`.

Ajouté dans la version 3.2.

rotate(*n=1*)

Décale les éléments de la *deque* de *n* places vers la droite (le dernier élément revient au début). Si *n* est négatif, décale vers la gauche.

Quand la *deque* n'est pas vide, un décalage d'une place vers la droite équivaut à `d.appendleft(d.pop())` et un décalage d'une place vers la gauche est équivalent à `d.append(d.popleft())`.

Les objets *deques* fournissent également un attribut en lecture seule :

maxlen

La taille maximale d'une *deque*, ou `None` si illimitée.

Ajouté dans la version 3.1.

In addition to the above, deques support iteration, pickling, `len(d)`, `reversed(d)`, `copy.copy(d)`, `copy.deepcopy(d)`, membership testing with the `in` operator, and subscript references such as `d[0]` to access the first element. Indexed access is $O(1)$ at both ends but slows to $O(n)$ in the middle. For fast random access, use lists instead.

Depuis la version 3.5, les *deques* gèrent `__add__()`, `__mul__()` et `__imul__()`.

Exemple :

```
>>> from collections import deque
>>> d = deque('ghi')                 # make a new deque with three items
>>> for elem in d:                   # iterate over the deque's elements
...     print(elem.upper())
G
H
I

>>> d.append('j')                    # add a new entry to the right side
>>> d.appendleft('f')                # add a new entry to the left side
>>> d                                # show the representation of the deque
deque(['f', 'g', 'h', 'i', 'j'])

>>> d.pop()                          # return and remove the rightmost item
'j'
>>> d.popleft()                      # return and remove the leftmost item
'f'
>>> list(d)                          # list the contents of the deque
['g', 'h', 'i']
>>> d[0]                             # peek at leftmost item
'g'
>>> d[-1]                            # peek at rightmost item
'i'

>>> list(reversed(d))                # list the contents of a deque in reverse
['i', 'h', 'g']
>>> 'h' in d                         # search the deque
True
>>> d.extend('jkl')                  # add multiple elements at once
>>> d
deque(['g', 'h', 'i', 'j', 'k', 'l'])
>>> d.rotate(1)                      # right rotation
>>> d
deque(['l', 'g', 'h', 'i', 'j', 'k'])
>>> d.rotate(-1)                     # left rotation
>>> d
deque(['g', 'h', 'i', 'j', 'k', 'l'])

>>> deque(reversed(d))               # make a new deque in reverse order
deque(['l', 'k', 'j', 'i', 'h', 'g'])
>>> d.clear()                        # empty the deque
>>> d.pop()                          # cannot pop from an empty deque
```

(suite sur la page suivante)

(suite de la page précédente)

```
Traceback (most recent call last):
    File "<pyshell#6>", line 1, in -toplevel-
        d.pop()
IndexError: pop from an empty deque

>>> d.extendleft('abc')              # extendleft() reverses the input order
>>> d
deque(['c', 'b', 'a'])
```

Cas pratiques utilisant `deque`

Cette partie montre diverses approches afin de travailler avec les *deques*.

Les *deques* à taille limitée apportent une fonctionnalité similaire au filtre `tail` d'Unix :

```
def tail(filename, n=10):
    'Return the last n lines of a file'
    with open(filename) as f:
        return deque(f, n)
```

Une autre approche d'utilisation des *deques* est de maintenir une séquence d'éléments récemment ajoutés en les ajoutant à droite et en retirant les anciens par la gauche :

```
def moving_average(iterable, n=3):
    # moving_average([40, 30, 50, 46, 39, 44]) --> 40.0 42.0 45.0 43.0
    # https://en.wikipedia.org/wiki/Moving_average
    it = iter(iterable)
    d = deque(itertools.islice(it, n-1))
    d.appendleft(0)
    s = sum(d)
    for elem in it:
        s += elem - d.popleft()
        d.append(elem)
        yield s / n
```

Un ordonnancement en round-robin peut être implémenté avec des entrées itérateurs stockées dans une `deque`. Les valeurs sont produites par l'itérateur actif en position zéro. Si cet itérateur est épuisé, il peut être retiré avec la méthode `popleft()` ; ou bien il peut être remis à la fin avec la méthode `rotate()` :

```
def roundrobin(*iterables):
    "roundrobin('ABC', 'D', 'EF') --> A D E B F C"
    iterators = deque(map(iter, iterables))
    while iterators:
        try:
            while True:
                yield next(iterators[0])
                iterators.rotate(-1)
        except StopIteration:
            # Remove an exhausted iterator.
            iterators.popleft()
```

La méthode `rotate()` apporte une façon d'implémenter la sélection d'intervalle (*slicing*) et les suppressions pour les `deques`. Par exemple, une implémentation de `del d[n]` en Python pur utilise la méthode `rotate()` pour mettre en position les éléments à éjecter :

```
def delete_nth(d, n):
    d.rotate(-n)
```

(suite sur la page suivante)

(suite de la page précédente)

```
    d.popleft()
    d.rotate(n)
```

Pour implémenter le *slicing* pour les `deques`, il est possible d'utiliser une approche similaire en appliquant `rotate()` afin d'apporter un élément cible à l'extrémité gauche de la *deque*. On éjecte les anciennes entrées avec `popleft()` et on ajoute les nouvelles avec `extend()`, puis on inverse la rotation. Il est aisé d'implémenter les manipulations des piles inspirées du Forth telles que `dup`, `drop`, `swap`, `over`, `pick`, `rot` et `roll`.

8.4.4 Objets `defaultdict`

class collections.**defaultdict**(*default_factory=None*, /[, ...])

Renvoie un nouvel objet qui se comporte comme un dictionnaire. `defaultdict` est une sous-classe de la classe native `dict`. Elle surcharge une méthode et ajoute une variable d'instance modifiable. Les autres fonctionnalités sont les mêmes que celles des objets `dict` et ne sont pas documentées ici.

Le premier argument fournit la valeur initiale de l'attribut `default_factory` qui doit être un objet appelable sans paramètre ou `None`, sa valeur par défaut. Tous les autres arguments sont traités comme si on les passait au constructeur de `dict`, y compris les arguments nommés.

En plus des opérations usuelles de `dict`, les objets `defaultdict` gèrent les méthodes supplémentaires suivantes :

__missing__(*key*)

Si l'attribut `default_factory` est `None`, lève une exception `KeyError` avec *key* comme argument.

Si `default_fatory` ne vaut pas `None`, cet attribut est appelé sans argument pour fournir une valeur par défaut pour la *key* demandée. Cette valeur est insérée dans le dictionnaire avec pour clé *key* et est renvoyée.

Si appeler `default_factory` lève une exception, celle-ci est transmise inchangée.

This method is called by the `__getitem__()` method of the `dict` class when the requested key is not found ; whatever it returns or raises is then returned or raised by `__getitem__()`.

Note that `__missing__()` is *not* called for any operations besides `__getitem__()`. This means that `get()` will, like normal dictionaries, return `None` as a default rather than using `default_factory`.

Les objets `defaultdict` gèrent la variable d'instance :

default_factory

Cet attribut est utilisé par la méthode `__missing__()` ; il est initialisé par le premier argument passé au constructeur, s'il est spécifié, sinon par `None`.

Modifié dans la version 3.9 : ajout des opérateurs fusion (`|`) et mise-à-jour (`|=`) tels que définis dans **PEP 584**.

Exemples utilisant `defaultdict`

Utiliser `list` comme `default_factory` facilite le regroupement d'une séquence de paires clé-valeur en un dictionnaire de listes :

```
>>> s = [('yellow', 1), ('blue', 2), ('yellow', 3), ('blue', 4), ('red', 1)]
>>> d = defaultdict(list)
>>> for k, v in s:
...     d[k].append(v)
...
>>> sorted(d.items())
[('blue', [2, 4]), ('red', [1]), ('yellow', [1, 3])]
```

When each key is encountered for the first time, it is not already in the mapping ; so an entry is automatically created using the `default_factory` function which returns an empty `list`. The `list.append()` operation then attaches the value to the new list. When keys are encountered again, the look-up proceeds normally (returning the list for that

key) and the `list.append()` operation adds another value to the list. This technique is simpler and faster than an equivalent technique using *dict.setdefault()* :

```
>>> d = {}
>>> for k, v in s:
...     d.setdefault(k, []).append(v)
...
>>> sorted(d.items())
[('blue', [2, 4]), ('red', [1]), ('yellow', [1, 3])]
```

Utiliser *int* comme *default_factory* rend la classe *defaultdict* pratique pour le comptage (comme un sac ou multi-ensemble dans d'autres langages) :

```
>>> s = 'mississippi'
>>> d = defaultdict(int)
>>> for k in s:
...     d[k] += 1
...
>>> sorted(d.items())
[('i', 4), ('m', 1), ('p', 2), ('s', 4)]
```

Quand une lettre est rencontrée pour la première fois, elle n'est pas dans le dictionnaire, donc la fonction *default_factory* appelle *int()* pour mettre un nouveau compteur à zéro. L'incrémentation augmente ensuite le comptage pour chaque lettre.

La fonction *int()* qui retourne toujours zéro est simplement une fonction constante particulière. Un moyen plus flexible et rapide de créer une fonction constante est d'utiliser une fonction lambda qui peut fournir n'importe quelle valeur constante (pas seulement zéro) :

```
>>> def constant_factory(value):
...     return lambda: value
...
>>> d = defaultdict(constant_factory('<missing>'))
>>> d.update(name='John', action='ran')
>>> '%(name)s %(action)s to %(object)s' % d
'John ran to <missing>'
```

Utiliser *set* comme *default_factory* rend la classe *defaultdict* pratique pour créer un dictionnaire d'ensembles :

```
>>> s = [('red', 1), ('blue', 2), ('red', 3), ('blue', 4), ('red', 1), ('blue', 4)]
>>> d = defaultdict(set)
>>> for k, v in s:
...     d[k].add(v)
...
>>> sorted(d.items())
[('blue', {2, 4}), ('red', {1, 3})]
```

8.4.5 `namedtuple()` : fonction de construction pour *n*-uplets avec des champs nommés

Les *n*-uplets nommés assignent une signification à chacun de leur élément, ce qui rend le code plus lisible et explicite. Ils peuvent être utilisés partout où les *n*-uplets natifs sont utilisés, et ils ajoutent la possibilité d'accéder à leurs champs grâce à leur nom au lieu de leur index de position.

`collections.`**`namedtuple`**(*typename*, *field_names*, *, *rename=False*, *defaults=None*, *module=None*)

Renvoie une nouvelle sous-classe de `tuple` appelée *typename*. Elle est utilisée pour créer des objets se comportant comme les *n*-uplets qui ont des champs accessibles par recherche d'attribut en plus d'être indexables

et itérables. Les instances de cette sous-classe possèdent aussi une *docstring* explicite (avec *type_name* et les *field_names*) et une méthode `__repr__()` pratique qui liste le contenu du *n*-uplet au format `nom=valeur`.

field_names peut être une séquence de chaînes de caractères telle que `['x', 'y']` ou bien une unique chaîne de caractères où les noms de champs sont séparés par un espace et/ou une virgule, par exemple `'x y'` ou `'x, y'`.

N'importe quel identifiant Python peut être utilisé pour un nom de champ hormis ceux commençant par un tiret bas. Les identifiants valides peuvent contenir des lettres, des chiffres (sauf en première position) et des tirets bas (sauf en première position). Un identifiant ne peut pas être un *keyword* tel que `class`, `for`, `return`, `global`, `pass` ou `raise`.

Si *rename* vaut `True`, alors les noms de champs invalides sont automatiquement renommés en noms positionnels. Par exemple, `['abc', 'def', 'ghi', 'abc']` est converti en `['abc, '_1', 'ghi', '_3']` afin d'éliminer le mot-clé `def` et le doublon de `abc`.

defaults peut être `None` ou un *iterable* de valeurs par défaut. Comme les champs avec une valeur par défaut doivent être définis après les champs sans valeur par défaut, les *defaults* sont appliqués aux paramètres les plus à droite. Par exemple, si les noms des champs sont `['x', 'y', 'z']` et les valeurs par défaut `(1, 2)`, alors `x` est un argument obligatoire tandis que `y` et `y` valent par défaut `1` et `2`.

If *module* is defined, the `__module__` attribute of the named tuple is set to that value.

Les instances de *n*-uplets nommés n'ont pas de dictionnaires propres, elles sont donc légères et ne requièrent pas plus de mémoire que les *n*-uplets natifs.

Pour permettre la sérialisation, la classe de *n*-uplet nommée doit être assignée à une variable qui correspond à *typename*.

Modifié dans la version 3.1 : Gestion de *rename*.

Modifié dans la version 3.6 : Les paramètres *verbose* et *rename* deviennent des *arguments obligatoirement nommés*.

Modifié dans la version 3.6 : Ajout du paramètre *module*.

Modifié dans la version 3.7 : Suppression du paramètre *verbose* et de l'attribut `_source`.

Modifié dans la version 3.7 : Ajout du paramètre *defaults* et de l'attribut `_field_defaults`.

```
>>> # Basic example
>>> Point = namedtuple('Point', ['x', 'y'])
>>> p = Point(11, y=22)     # instantiate with positional or keyword arguments
>>> p[0] + p[1]             # indexable like the plain tuple (11, 22)
33
>>> x, y = p                # unpack like a regular tuple
>>> x, y
(11, 22)
>>> p.x + p.y               # fields also accessible by name
33
>>> p                       # readable __repr__ with a name=value style
Point(x=11, y=22)
```

Les *n*-uplets nommés sont particulièrement utiles pour associer des noms de champs à des *n*-uplets renvoyés par les modules *csv* ou *sqlite3* :

```
EmployeeRecord = namedtuple('EmployeeRecord', 'name, age, title, department,␣
↪paygrade')

import csv
for emp in map(EmployeeRecord._make, csv.reader(open("employees.csv", "rb"))):
    print(emp.name, emp.title)

import sqlite3
```

(suite sur la page suivante)

(suite de la page précédente)

```
conn = sqlite3.connect('/companydata')
cursor = conn.cursor()
cursor.execute('SELECT name, age, title, department, paygrade FROM employees')
for emp in map(EmployeeRecord._make, cursor.fetchall()):
    print(emp.name, emp.title)
```

En plus des méthodes héritées de `tuple`, les *n*-uplets nommés implémentent trois méthodes et deux attributs supplémentaires. Pour éviter les conflits avec noms de champs, leurs noms commencent par un tiret bas.

classmethod `somenamedtuple.`**`_make`**`(`*iterable*`)`

Méthode de classe qui construit une nouvelle instance à partir d'une séquence ou d'un itérable existant.

```
>>> t = [11, 22]
>>> Point._make(t)
Point(x=11, y=22)
```

`somenamedtuple.`**`_asdict`**`()`

Renvoie un nouveau `dict` qui associe chaque nom de champ à sa valeur correspondante :

```
>>> p = Point(x=11, y=22)
>>> p._asdict()
{'x': 11, 'y': 22}
```

Modifié dans la version 3.1 : Renvoie un `OrderedDict` au lieu d'un `dict` natif.

Modifié dans la version 3.8 : renvoie un `dict` natif plutôt qu'un `OrderedDict`. À partir de Python 3.7, les dictionnaires natifs garantissent la préservation de l'ordre. Si les autres fonctionnalités d'`OrderedDict` sont nécessaires, la solution préconisée est de convertir le résultat vers le type souhaité : `OrderedDict(nt._asdict())`.

`somenamedtuple.`**`_replace`**`(`***kwargs*`)`

Renvoie une nouvelle instance du *n*-uplet nommé en remplaçant les champs spécifiés par leurs nouvelles valeurs :

```
>>> p = Point(x=11, y=22)
>>> p._replace(x=33)
Point(x=33, y=22)

>>> for partnum, record in inventory.items():
...     inventory[partnum] = record._replace(price=newprices[partnum],␣
↪timestamp=time.now())
```

Named tuples are also supported by generic function `copy.replace()`.

Modifié dans la version 3.13 : Raise `TypeError` instead of `ValueError` for invalid keyword arguments.

`somenamedtuple.`**`_fields`**

Tuple de chaînes de caractères listant les noms de champs. Pratique pour l'introspection et pour créer de nouveaux types de *n*-uplets nommés à partir d'existants.

```
>>> p._fields            # view the field names
('x', 'y')

>>> Color = namedtuple('Color', 'red green blue')
>>> Pixel = namedtuple('Pixel', Point._fields + Color._fields)
>>> Pixel(11, 22, 128, 255, 0)
Pixel(x=11, y=22, red=128, green=255, blue=0)
```

somenamedtuple.**_field_defaults**

Dictionnaire qui assigne les valeurs par défaut aux noms des champs.

```
>>> Account = namedtuple('Account', ['type', 'balance'], defaults=[0])
>>> Account._field_defaults
{'balance': 0}
>>> Account('premium')
Account(type='premium', balance=0)
```

Pour récupérer un champ dont le nom est une chaîne de caractères, utilisez la fonction `getattr()` :

```
>>> getattr(p, 'x')
11
```

Pour convertir un dictionnaire en *n*-uplet nommé, utilisez l'opérateur double-étoile (comme expliqué dans tut-unpacking-arguments) :

```
>>> d = {'x': 11, 'y': 22}
>>> Point(**d)
Point(x=11, y=22)
```

Il est aisé d'ajouter ou de modifier les fonctionnalités des *n*-uplets nommés grâce à l'héritage puisqu'il s'agit de simples classes. Voici comment ajouter un champ calculé avec une longueur fixe d'affichage :

```
>>> class Point(namedtuple('Point', ['x', 'y'])):
...     __slots__ = ()
...     @property
...     def hypot(self):
...         return (self.x ** 2 + self.y ** 2) ** 0.5
...     def __str__(self):
...         return 'Point: x=%6.3f  y=%6.3f  hypot=%6.3f' % (self.x, self.y, self.
↪hypot)

>>> for p in Point(3, 4), Point(14, 5/7):
...     print(p)
Point: x= 3.000  y= 4.000  hypot= 5.000
Point: x=14.000  y= 0.714  hypot=14.018
```

La sous-classe ci-dessus définit `__slots__` comme un *n*-uplet vide. Cela permet de garder une emprunte mémoire faible en empêchant la création de dictionnaire d'instance.

L'héritage n'est pas pertinent pour ajouter de nouveaux champs. Il est préférable de simplement créer un nouveau type de *n*-uplet nommé avec l'attribut `_fields` :

```
>>> Point3D = namedtuple('Point3D', Point._fields + ('z',))
```

Les *docstrings* peuvent être personnalisées en modifiant directement l'attribut `__doc__` :

```
>>> Book = namedtuple('Book', ['id', 'title', 'authors'])
>>> Book.__doc__ += ': Hardcover book in active collection'
>>> Book.id.__doc__ = '13-digit ISBN'
>>> Book.title.__doc__ = 'Title of first printing'
>>> Book.authors.__doc__ = 'List of authors sorted by last name'
```

Modifié dans la version 3.5 : La propriété devient éditable.

Voir aussi

— Voir `typing.NamedTuple()` pour un moyen d'ajouter des indications de type pour les *n*-uplets nommés. Cela propose aussi une notation élégante utilisant le mot-clé `class` :

```
class Component(NamedTuple):
    part_number: int
    weight: float
    description: Optional[str] = None
```

— Voir `types.SimpleNamespace()` pour un espace de nommage mutable basé sur un dictionnaire sous-jacent à la place d'un *n*-uplet.

— Le module `dataclasses` fournit un décorateur et des fonctions pour ajouter automatiquement des méthodes spéciales générées aux classes définies par l'utilisateur.

8.4.6 Objets `OrderedDict`

Les dictionnaires ordonnés sont des dictionnaires comme les autres mais possèdent des capacités supplémentaires pour s'ordonner. Ils sont maintenant moins importants puisque la classe native `dict` sait se souvenir de l'ordre d'insertion (cette fonctionnalité a été garantie par Python 3.7).

Quelques différences persistent vis-à-vis de `dict` :

— Les `dict` classiques ont été conçus pour être performants dans les opérations de correspondance. Garder une trace de l'ordre d'insertion était secondaire.

— Les `OrderedDict` ont été conçus pour être performants dans les opérations de ré-arrangement. L'occupation mémoire, la vitesse de parcours et les performances de mise à jour étaient secondaires.

— The `OrderedDict` algorithm can handle frequent reordering operations better than `dict`. As shown in the recipes below, this makes it suitable for implementing various kinds of LRU caches.

— Le test d'égalité de `OrderedDict` vérifie si l'ordre correspond.

A regular `dict` can emulate the order sensitive equality test with `p == q and all(k1 == k2 for k1, k2 in zip(p, q))`.

— La méthode `popitem()` de `OrderedDict` possède une signature différente. Elle accepte un argument optionnel pour spécifier quel élément doit être enlevé.

A regular `dict` can emulate OrderedDict's `od.popitem(last=True)` with `d.popitem()` which is guaranteed to pop the rightmost (last) item.

A regular `dict` can emulate OrderedDict's `od.popitem(last=False)` with `(k := next(iter(d)), d.pop(k))` which will return and remove the leftmost (first) item if it exists.

— `OrderedDict` possède une méthode `move_to_end()` pour déplacer efficacement un élément à la fin.

A regular `dict` can emulate OrderedDict's `od.move_to_end(k, last=True)` with `d[k] = d.pop(k)` which will move the key and its associated value to the rightmost (last) position.

A regular `dict` does not have an efficient equivalent for OrderedDict's `od.move_to_end(k, last=False)` which moves the key and its associated value to the leftmost (first) position.

— Avant Python 3.8, `dict` n'a pas de méthode `__reversed__()`.

class `collections.`**`OrderedDict`**(*[items]*)

Renvoie une instance d'une sous-classe de `dict` qui possède des méthodes spécialisées pour redéfinir l'ordre du dictionnaire.

Ajouté dans la version 3.1.

popitem(*last=True*)

La méthode `popitem()` pour les dictionnaires ordonnés retire et renvoie une paire `(clé, valeur)`. Les paires sont renvoyées comme pour une pile, c'est-à-dire dernier entré, premier sorti (en anglais LIFO) si *last* vaut `True`. Si *last* vaut `False`, alors les paires sont renvoyées comme pour une file, c'est-à-dire premier entré, premier sorti (en anglais FIFO (first-in, first-out)).

move_to_end(*key*, *last=True*)

Move an existing *key* to either end of an ordered dictionary. The item is moved to the right end if *last* is true (the default) or to the beginning if *last* is false. Raises `KeyError` if the *key* does not exist :

```
>>> d = OrderedDict.fromkeys('abcde')
>>> d.move_to_end('b')
>>> ''.join(d)
'acdeb'
>>> d.move_to_end('b', last=False)
>>> ''.join(d)
'bacde'
```

Ajouté dans la version 3.2.

En plus des méthodes usuelles des dictionnaires, les dictionnaires ordonnés gèrent l'itération en sens inverse grâce à `reversed()`.

Equality tests between `OrderedDict` objects are order-sensitive and are roughly equivalent to `list(od1.items())==list(od2.items())`.

Equality tests between `OrderedDict` objects and other `Mapping` objects are order-insensitive like regular dictionaries. This allows `OrderedDict` objects to be substituted anywhere a regular dictionary is used.

Modifié dans la version 3.5 : Les *vues* d'éléments, de clés et de valeurs de `OrderedDict` gèrent maintenant l'itération en sens inverse en utilisant `reversed()`.

Modifié dans la version 3.6 : Suite à l'acceptation de la **PEP 468**, l'ordre des arguments nommés passés au constructeur et à la méthode `update()` de `OrderedDict` est conservé.

Modifié dans la version 3.9 : ajout des opérateurs fusion (`|`) et mise-à-jour (`|=`) tels que définis dans **PEP 584**.

Exemples et cas pratiques utilisant `OrderDict`

Il est facile de créer une variante de dictionnaire ordonné qui retient l'ordre dans lequel les clés ont été insérées *en dernier*. Si une nouvelle entrée écrase une existante, la position d'insertion d'origine est modifiée et déplacée à la fin :

```
class LastUpdatedOrderedDict(OrderedDict):
    'Store items in the order the keys were last added'

    def __setitem__(self, key, value):
        super().__setitem__(key, value)
        self.move_to_end(key)
```

Un `OrderedDict` peut aussi être utile pour implémenter des variantes de `functools.lru_cache()` :

```
from collections import OrderedDict
from time import time

class TimeBoundedLRU:
    "LRU Cache that invalidates and refreshes old entries."

    def __init__(self, func, maxsize=128, maxage=30):
        self.cache = OrderedDict()      # { args : (timestamp, result)}
        self.func = func
        self.maxsize = maxsize
        self.maxage = maxage

    def __call__(self, *args):
        if args in self.cache:
            self.cache.move_to_end(args)
            timestamp, result = self.cache[args]
            if time() - timestamp <= self.maxage:
                return result
        result = self.func(*args)
```

(suite sur la page suivante)

(suite de la page précédente)

```
        self.cache[args] = time(), result
        if len(self.cache) > self.maxsize:
            self.cache.popitem(last=False)
        return result
```

```
class MultiHitLRUCache:
    """ LRU cache that defers caching a result until
        it has been requested multiple times.

        To avoid flushing the LRU cache with one-time requests,
        we don't cache until a request has been made more than once.

    """

    def __init__(self, func, maxsize=128, maxrequests=4096, cache_after=1):
        self.requests = OrderedDict()    # { uncached_key : request_count }
        self.cache = OrderedDict()       # { cached_key : function_result }
        self.func = func
        self.maxrequests = maxrequests   # max number of uncached requests
        self.maxsize = maxsize           # max number of stored return values
        self.cache_after = cache_after

    def __call__(self, *args):
        if args in self.cache:
            self.cache.move_to_end(args)
            return self.cache[args]
        result = self.func(*args)
        self.requests[args] = self.requests.get(args, 0) + 1
        if self.requests[args] <= self.cache_after:
            self.requests.move_to_end(args)
            if len(self.requests) > self.maxrequests:
                self.requests.popitem(last=False)
        else:
            self.requests.pop(args, None)
            self.cache[args] = result
            if len(self.cache) > self.maxsize:
                self.cache.popitem(last=False)
        return result
```

8.4.7 Objets `UserDict`

La classe `UserDict` se comporte comme une surcouche autour des dictionnaires. L'utilité de cette classe est réduite, car on peut maintenant hériter directement de `dict`. Cependant, il peut être plus facile de travailler avec celle-ci, car le dictionnaire sous-jacent est accessible comme attribut.

class collections.**UserDict**([*initialdata*])

Classe simulant un dictionnaire. Les instances de `UserDict` possèdent un attribut `data` où est stocké leur contenu sous forme de dictionnaire natif. Si *initialdata* est spécifié, alors `data` est initialisé avec son contenu. Remarquez qu'une référence vers *initialdata* n'est pas conservée, ce qui permet de l'utiliser pour d'autres tâches.

En plus de gérer les méthodes et opérations des dictionnaires, les instances de `UserDict` fournissent l'attribut suivant :

data

Un dictionnaire natif où est stocké le contenu de la classe `UserDict`.

8.4.8 Objets `UserList`

Cette classe agit comme une surcouche autour des objets `list`. C'est une classe mère utile pour vos classes listes-compatibles qui peuvent en hériter et surcharger les méthodes existantes ou en ajouter de nouvelles. Ainsi, on peut ajouter de nouveaux comportements aux listes.

L'utilité de cette classe a été partiellement réduite par la possibilité d'hériter directement de `list`. Cependant, il peut être plus facile de travailler avec cette classe, car la liste sous-jacente est accessible via un attribut.

class collections.**UserList**([*list*])

Classe simulant une liste. Les instances de `UserList` possèdent un attribut `UserList` où est stocké leur contenu sous forme de liste native. Il est initialement une copie de *list*, ou `[]` par défaut. *list* peut être un itérable, par exemple une liste native ou un objet `UserList`.

En plus de gérer les méthodes et opérations des séquences mutables, les instances de `UserList` possèdent l'attribut suivant :

data

Un objet `list` natif utilisé pour stocker le contenu de la classe `UserList`.

Prérequis pour l'héritage : Les sous-classes de `UserList` doivent implémenter un constructeur qui peut être appelé avec zéro ou un argument. Les opérations sur les listes qui renvoient une nouvelle séquence essayent de créer une instance de la classe courante. C'est pour cela que le constructeur doit pouvoir être appelé avec un unique paramètre, un objet séquence utilisé comme source de données.

Si une classe fille ne remplit pas cette condition, toutes les méthodes spéciales gérées par cette classe devront être implémentées à nouveau. Merci de consulter les sources pour obtenir des informations sur les méthodes qui doivent être fournies dans ce cas.

8.4.9 Objets `UserString`

La classe `UserString` agit comme une surcouche autour des objets `str`. L'utilité de cette classe a été partiellement réduite par la possibilité d'hériter directement de `str`. Cependant, il peut être plus facile de travailler avec cette classe, car la chaîne de caractère sous-jacente est accessible via un attribut.

class collections.**UserString**(*seq*)

Classe simulant une chaîne de caractères. Les instances de `UserString` possèdent un attribut `data` où est stocké leur contenu sous forme de chaîne de caractères native. Le contenu de l'instance est initialement une copie de *seq*, qui peut être n'importe quel objet convertible en chaîne de caractère avec la fonction native `str()`.

En plus de gérer les méthodes et opérations sur les chaînes de caractères, les instances de `UserString` possèdent l'attribut suivant :

data

Un objet `str` natif utilisé pour stocker le contenu de la classe `UserString`.

Modifié dans la version 3.5 : Nouvelles méthodes `__getnewargs__`, `__rmod__`, `casefold`, `format_map`, `isprintable` et `maketrans`.

8.5 `collections.abc` --- Abstract Base Classes for Containers

Ajouté dans la version 3.3 : Auparavant, ce module faisait partie du module `collections`.

Code source : Lib/_collections_abc.py

This module provides *abstract base classes* that can be used to test whether a class provides a particular interface ; for example, whether it is *hashable* or whether it is a *mapping*.

An `issubclass()` or `isinstance()` test for an interface works in one of three ways.

1) A newly written class can inherit directly from one of the abstract base classes. The class must supply the required abstract methods. The remaining mixin methods come from inheritance and can be overridden if desired. Other methods may be added as needed :

```
class C(Sequence):                          # Direct inheritance
    def __init__(self): ...                 # Extra method not required by the ABC
    def __getitem__(self, index):  ...      # Required abstract method
    def __len__(self):  ...                 # Required abstract method
    def count(self, value): ...             # Optionally override a mixin method
```

```
>>> issubclass(C, Sequence)
True
>>> isinstance(C(), Sequence)
True
```

2) Existing classes and built-in classes can be registered as "virtual subclasses" of the ABCs. Those classes should define the full API including all of the abstract methods and all of the mixin methods. This lets users rely on `issubclass()` or `isinstance()` tests to determine whether the full interface is supported. The exception to this rule is for methods that are automatically inferred from the rest of the API :

```
class D:                                     # No inheritance
    def __init__(self): ...                  # Extra method not required by the ABC
    def __getitem__(self, index):  ...       # Abstract method
    def __len__(self):  ...                  # Abstract method
    def count(self, value): ...              # Mixin method
    def index(self, value): ...              # Mixin method

Sequence.register(D)                         # Register instead of inherit
```

```
>>> issubclass(D, Sequence)
True
>>> isinstance(D(), Sequence)
True
```

In this example, class `D` does not need to define `__contains__`, `__iter__`, and `__reversed__` because the in-operator, the *iteration* logic, and the `reversed()` function automatically fall back to using `__getitem__` and `__len__`.

3) Some simple interfaces are directly recognizable by the presence of the required methods (unless those methods have been set to `None`) :

```
class E:
    def __iter__(self): ...
    def __next__(self): ...
```

```
>>> issubclass(E, Iterable)
True
>>> isinstance(E(), Iterable)
True
```

Complex interfaces do not support this last technique because an interface is more than just the presence of method names. Interfaces specify semantics and relationships between methods that cannot be inferred solely from the presence of specific method names. For example, knowing that a class supplies `__getitem__`, `__len__`, and `__iter__` is insufficient for distinguishing a `Sequence` from a `Mapping`.

Ajouté dans la version 3.9 : These abstract classes now support `[]`. See *Type Alias générique* and **PEP 585**.

8.5.1 Classes de base abstraites de collections

Le module collections apporte les *ABC* suivantes :

ABC	Hérite de	Méthodes abstraites	Méthodes *mixin*
`Container`[1]		`__contains__`	
`Hashable`[1]		`__hash__`	
`Iterable`[1 2]		`__iter__`	
`Iterator`[1]	`Iterable`	`__next__`	`__iter__`
`Reversible`[1]	`Iterable`	`__reversed__`	
`Generator`[1]	`Iterator`	`send`, `throw`	`close`, `__iter__`, `__next__`
`Sized`[1]		`__len__`	
`Callable`[1]		`__call__`	
`Collection`[1]	`Sized`, `Iterable`, `Container`	`__contains__`, `__iter__`, `__len__`	
`Sequence`	`Reversibl` `Collectio`	`__getitem__`, `__len__`	`__contains__`, `__iter__`, `__reversed__`, `index` et `count`
`MutableSequence`	`Sequence`	`__getitem__`, `__setitem__`, `__delitem__`, `__len__`, `insert`	Inherited `Sequence` methods and `append`, `clear`, `reverse`, `extend`, `pop`, `remove`, and `__iadd__`
`Set`	`Collectio`	`__contains__`, `__iter__`, `__len__`	`__le__`, `__lt__`, `__eq__`, `__ne__`, `__gt__`, `__ge__`, `__and__`, `__or__`, `__sub__`, `__rsub__`, `__xor__`, `__rxor__` and `isdisjoint`
`MutableSet`	`Set`	`__contains__`, `__iter__`, `__len__`, `add`, `discard`	Méthodes héritées de `Set`, et `clear`, `pop`, `remove`, `__ior__`, `__iand__`, `__ixor__` et `__isub__`
`Mapping`	`Collectio`	`__getitem__`, `__iter__`, `__len__`	`__contains__`, `keys`, `items`, `values`, `get`, `__eq__` et `__ne__`
`MutableMapping`	`Mapping`	`__getitem__`, `__setitem__`, `__delitem__`, `__iter__`, `__len__`	Méthodes héritées de `Mapping`, et `pop`, `popitem`, `clear`, `update` et `setdefault`
`MappingView`	`Sized`		`__init__`, `__len__` and `__repr__`
`ItemsView`	`MappingVi` `Set`		`__contains__`, `__iter__`
`KeysView`	`MappingVi` `Set`		`__contains__`, `__iter__`
`ValuesView`	`MappingVi` `Collectio`		`__contains__`, `__iter__`
`Awaitable`[1]		`__await__`	
`Coroutine`[1]	`Awaitable`	`send`, `throw`	`close`
`AsyncIterable`[1]		`__aiter__`	
`AsyncIterator`[1]	`AsyncIter`	`__anext__`	`__aiter__`
`AsyncGenerator`[1]	`AsyncIter`	`asend`, `athrow`	`aclose`, `__aiter__`, `__anext__`
`Buffer`[1]		`__buffer__`	

1. These ABCs override `__subclasshook__()` to support testing an interface by verifying the required methods are present and have not been set to `None`. This only works for simple interfaces. More complex interfaces require registration or direct subclassing.
2. Checking `isinstance(obj, Iterable)` detects classes that are registered as `Iterable` or that have an `__iter__()` method, but it does not detect classes that iterate with the `__getitem__()` method. The only reliable way to determine whether an object is *iterable* is to call `iter(obj)`.

Notes

8.5.2 Collections Abstract Base Classes -- Detailed Descriptions

class collections.abc.**Container**

ABC for classes that provide the `__contains__()` method.

class collections.abc.**Hashable**

ABC for classes that provide the `__hash__()` method.

class collections.abc.**Sized**

ABC for classes that provide the `__len__()` method.

class collections.abc.**Callable**

ABC for classes that provide the `__call__()` method.

See *Annotating callable objects* for details on how to use `Callable` in type annotations.

class collections.abc.**Iterable**

ABC for classes that provide the *`__iter__()`* method.

Checking `isinstance(obj, Iterable)` detects classes that are registered as *`Iterable`* or that have an *`__iter__()`* method, but it does not detect classes that iterate with the `__getitem__()` method. The only reliable way to determine whether an object is *iterable* is to call `iter(obj)`.

class collections.abc.**Collection**

ABC pour les classes de conteneurs itérables et *sized*.

Ajouté dans la version 3.6.

class collections.abc.**Iterator**

ABC pour les classes qui définissent les méthodes *`__iter__()`* et *`__next__()`*. Voir aussi la définition d'*itérateur*.

class collections.abc.**Reversible**

ABC for iterable classes that also provide the `__reversed__()` method.

Ajouté dans la version 3.6.

class collections.abc.**Generator**

ABC for *generator* classes that implement the protocol defined in **PEP 342** that extends *iterators* with the `send()`, `throw()` and `close()` methods.

See *Annotating generators and coroutines* for details on using `Generator` in type annotations.

Ajouté dans la version 3.5.

class collections.abc.**Sequence**

class collections.abc.**MutableSequence**

ABC pour les *séquences* immuables et mutables.

Implementation note : Some of the mixin methods, such as *`__iter__()`*, `__reversed__()` and `index()`, make repeated calls to the underlying `__getitem__()` method. Consequently, if `__getitem__()` is implemented with constant access speed, the mixin methods will have linear performance ; however, if the underlying method is linear (as it would be with a linked list), the mixins will have quadratic performance and will likely need to be overridden.

Modifié dans la version 3.5 : La méthode index() a ajouté le support des arguments *start* et *stop*.

class collections.abc.**Set**

class collections.abc.**MutableSet**

ABCs for read-only and mutable *sets*.

class collections.abc.**Mapping**

class collections.abc.**MutableMapping**

ABC pour les *tables de correspondances* immuables et mutables.

class collections.abc.**MappingView**

class collections.abc.**ItemsView**

class collections.abc.**KeysView**

class collections.abc.**ValuesView**

ABC pour les *vues* de *mappings* (tableaux de correspondances), d'éléments, de clés et de valeurs.

class collections.abc.**Awaitable**

ABC for *awaitable* objects, which can be used in `await` expressions. Custom implementations must provide the `__await__()` method.

Les objets *coroutines* et les instances de l'ABC `Coroutine` sont tous des instances de cette ABC.

> **Note**
>
> In CPython, generator-based coroutines (*generators* decorated with `@types.coroutine`) are *awaitables*, even though they do not have an `__await__()` method. Using `isinstance(gencoro, Awaitable)` for them will return `False`. Use `inspect.isawaitable()` to detect them.

Ajouté dans la version 3.5.

class collections.abc.**Coroutine**

ABC for *coroutine* compatible classes. These implement the following methods, defined in coroutine-objects : `send()`, `throw()`, and `close()`. Custom implementations must also implement `__await__()`. All `Coroutine` instances are also instances of `Awaitable`.

> **Note**
>
> In CPython, generator-based coroutines (*generators* decorated with `@types.coroutine`) are *awaitables*, even though they do not have an `__await__()` method. Using `isinstance(gencoro, Coroutine)` for them will return `False`. Use `inspect.isawaitable()` to detect them.

See *Annotating generators and coroutines* for details on using `Coroutine` in type annotations. The variance and order of type parameters correspond to those of `Generator`.

Ajouté dans la version 3.5.

class collections.abc.**AsyncIterable**

ABC for classes that provide an `__aiter__` method. See also the definition of *asynchronous iterable*.

Ajouté dans la version 3.5.

class collections.abc.**AsyncIterator**

ABC pour les classes qui définissent les méthodes `__aiter__` et `__anext__`. Voir aussi la définition d'*itérateur asynchrone*.

Ajouté dans la version 3.5.

class collections.abc.**AsyncGenerator**

ABC for *asynchronous generator* classes that implement the protocol defined in **PEP 525** and **PEP 492**.

See *Annotating generators and coroutines* for details on using `AsyncGenerator` in type annotations.

Ajouté dans la version 3.6.

class collections.abc.**Buffer**

ABC for classes that provide the `__buffer__()` method, implementing the buffer protocol. See **PEP 688**.

Ajouté dans la version 3.12.

8.5.3 Exemples et recettes

ABCs allow us to ask classes or instances if they provide particular functionality, for example :

```
size = None
if isinstance(myvar, collections.abc.Sized):
    size = len(myvar)
```

Several of the ABCs are also useful as mixins that make it easier to develop classes supporting container APIs. For example, to write a class supporting the full *Set* API, it is only necessary to supply the three underlying abstract methods : `__contains__()`, `__iter__()`, and `__len__()`. The ABC supplies the remaining methods such as `__and__()` and *isdisjoint()* :

```
class ListBasedSet(collections.abc.Set):
    ''' Alternate set implementation favoring space over speed
        and not requiring the set elements to be hashable. '''
    def __init__(self, iterable):
        self.elements = lst = []
        for value in iterable:
            if value not in lst:
                lst.append(value)

    def __iter__(self):
        return iter(self.elements)

    def __contains__(self, value):
        return value in self.elements

    def __len__(self):
        return len(self.elements)

s1 = ListBasedSet('abcdef')
s2 = ListBasedSet('defghi')
overlap = s1 & s2            # The __and__() method is supported automatically
```

Notes à propos de l'utilisation de *Set* et *MutableSet* comme *mixin* :

(1) Since some set operations create new sets, the default mixin methods need a way to create new instances from an *iterable*. The class constructor is assumed to have a signature in the form `ClassName(iterable)`. That assumption is factored-out to an internal *classmethod* called `_from_iterable()` which calls `cls(iterable)` to produce a new set. If the *Set* mixin is being used in a class with a different constructor signature, you will need to override `_from_iterable()` with a classmethod or regular method that can construct new instances from an iterable argument.

(2) To override the comparisons (presumably for speed, as the semantics are fixed), redefine `__le__()` and `__ge__()`, then the other operations will automatically follow suit.

(3) The *Set* mixin provides a `_hash()` method to compute a hash value for the set ; however, `__hash__()` is not defined because not all sets are *hashable* or immutable. To add set hashability using mixins, inherit from both *Set()* and *Hashable()*, then define `__hash__ = Set._hash`.

Voir aussi

— OrderedSet recipe pour un exemple construit sur *MutableSet*.
— Pour plus d'informations à propos des ABC, voir le module *abc* et la **PEP 3119**.

8.6 heapq --- Heap queue algorithm

Code source : Lib/heapq.py

Ce module expose une implémentation de l'algorithme de file de priorité, basée sur un tas.

Heaps are binary trees for which every parent node has a value less than or equal to any of its children. We refer to this condition as the heap invariant.

This implementation uses arrays for which `heap[k] <= heap[2*k+1]` and `heap[k] <= heap[2*k+2]` for all *k*, counting elements from zero. For the sake of comparison, non-existing elements are considered to be infinite. The interesting property of a heap is that its smallest element is always the root, `heap[0]`.

L'API ci-dessous diffère de la file de priorité classique par deux aspects : (a) l'indiçage commence à zéro. Cela complexifie légèrement la relation entre l'indice d'un nœud et les indices de ses fils mais est alignée avec l'indiçage commençant à zéro que Python utilise. (b) La méthode *pop* renvoie le plus petit élément et non le plus grand (appelé « tas-min » dans les manuels scolaires ; le « tas-max » étant généralement plus courant dans la littérature car il permet le classement sans tampon).

Ces deux points permettent d'aborder le tas comme une liste Python standard sans surprise : `heap[0]` est le plus petit élément et `heap.sort()` conserve l'invariant du tas !

Pour créer un tas, utilisez une liste initialisée à `[]` ou bien utilisez une liste existante et transformez la en tas à l'aide de la fonction `heapify()`.

Les fonctions suivantes sont fournies :

`heapq.`**`heappush`**(*heap*, *item*)

Introduit la valeur *item* dans le tas *heap*, en conservant l'invariance du tas.

`heapq.`**`heappop`**(*heap*)

Extraie le plus petit élément de *heap* en préservant l'invariant du tas. Si le tas est vide, une exception `IndexError` est levée. Pour accéder au plus petit élément sans le retirer, utilisez `heap[0]`.

`heapq.`**`heappushpop`**(*heap*, *item*)

Introduit l'élément *item* dans le tas, puis extraie le plus petit élément de *heap*. Cette action combinée est plus efficace que `heappush()` suivie par un appel séparé à `heappop()`.

`heapq.`**`heapify`**(*x*)

Transforme une liste *x* en un tas, sans utiliser de tampon et en temps linéaire.

`heapq.`**`heapreplace`**(*heap*, *item*)

Extraie le plus petit élément de *heap* et introduit le nouvel élément *item*. La taille du tas ne change pas. Si le tas est vide, une exception `IndexError` est levée.

Cette opération en une étape est plus efficace qu'un appel à `heappop()` suivi d'un appel à `heappush()` et est plus appropriée lorsque le tas est de taille fixe. La combinaison *pop*/*push* renvoie toujours un élément du tas et le remplace par *item*.

La valeur renvoyée peut être plus grande que l'élément *item* ajouté. Si cela n'est pas souhaitable, utilisez plutôt `heappushpop()` à la place. Sa combinaison *push*/*pop* renvoie le plus petit élément des deux valeurs et laisse la plus grande sur le tas.

Ce module contient également trois fonctions génériques utilisant les tas.

`heapq.`**`merge`**(**iterables*, *key=None*, *reverse=False*)

Fusionne plusieurs entrées ordonnées en une unique sortie ordonnée (par exemple, fusionne des entrées datées provenant de multiples journaux applicatifs). Renvoie un *iterator* sur les valeurs ordonnées.

Similaire à `sorted(itertools.chain(*iterables))` mais renvoie un itérable, ne stocke pas toutes les données en mémoire en une fois et suppose que chaque flux d'entrée est déjà classé (en ordre croissant).

A deux arguments optionnels qui doivent être fournis par mot clef.

key spécifie une *key function* d'un argument utilisée pour extraire une clef de comparaison de chaque élément de la liste. La valeur par défaut est `None` (compare les éléments directement).

reverse est une valeur booléenne. Si elle est `True`, la liste d'éléments est fusionnée comme si toutes les comparaisons étaient inversées. Pour obtenir un comportement similaire à `sorted(itertools.chain(*iterables), reverse=True)`, tous les itérables doivent être classés par ordre décroissant.

Modifié dans la version 3.5 : Ajout des paramètres optionnels *key* et *reverse*.

`heapq.`**`nlargest`**(*n*, *iterable*, *key=None*)

Renvoie une liste contenant les *n* plus grands éléments du jeu de données défini par *iterable*. Si l'option *key* est fournie, celle-ci spécifie une fonction à un argument qui est utilisée pour extraire la clé de comparaison de chaque élément dans *iterable* (par exemple, `key=str.lower`). Équivalent à : `sorted(iterable, key=key, reverse=True)[:n]`.

`heapq.`**`nsmallest`**(*n*, *iterable*, *key=None*)

Renvoie une liste contenant les *n* plus petits éléments du jeu de données défini par *iterable*. Si l'option *key* est fournie, celle-ci spécifie une fonction à un argument qui est utilisée pour extraire la clé de comparaison de chaque élément dans *iterable* (par exemple, `key=str.lower`). Équivalent à : `sorted(iterable, key=key)[:n]`.

Les deux fonctions précédentes sont les plus efficaces pour des petites valeurs de *n*. Pour de grandes valeurs, il est préférable d'utiliser la fonction `sorted()`. En outre, lorsque `n==1`, il est plus efficace d'utiliser les fonctions natives `min()` et `max()`. Si vous devez utiliser ces fonctions de façon répétée, il est préférable de transformer l'itérable en tas.

8.6.1 Exemples simples

Un tri par tas peut être implémenté en introduisant toutes les valeurs dans un tas puis en effectuant l'extraction des éléments un par un :

```
>>> def heapsort(iterable):
...     h = []
...     for value in iterable:
...         heappush(h, value)
...     return [heappop(h) for i in range(len(h))]
...
>>> heapsort([1, 3, 5, 7, 9, 2, 4, 6, 8, 0])
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Ceci est similaire à `sorted(iterable)` mais, contrairement à `sorted()`, cette implémentation n'est pas stable.

Les éléments d'un tas peuvent être des *n*-uplets. C'est pratique pour assigner des valeurs de comparaison (par exemple, des priorités de tâches) en plus de l'élément qui est suivi :

```
>>> h = []
>>> heappush(h, (5, 'write code'))
>>> heappush(h, (7, 'release product'))
>>> heappush(h, (1, 'write spec'))
>>> heappush(h, (3, 'create tests'))
>>> heappop(h)
(1, 'write spec')
```

8.6.2 Notes d'implémentation de la file de priorité

Une file de priorité est une application courante des tas et présente plusieurs défis d'implémentation :

- Stabilité du classement : comment s'assurer que deux tâches avec la même priorité sont renvoyées dans l'ordre de leur ajout ?
- La comparaison des couples (priorité, tâche) échoue si les priorités sont identiques et que les tâches n'ont pas de relation d'ordre par défaut.
- Si la priorité d'une tâche change, comment la déplacer à sa nouvelle position dans le tas ?

— Si une tâche en attente doit être supprimée, comment la trouver et la supprimer de la file ?

Une solution aux deux premiers problèmes consiste à stocker les entrées sous forme de liste à 3 éléments incluant la priorité, le numéro d'ajout et la tâche. Le numéro d'ajout sert à briser les égalités de telle sorte que deux tâches avec la même priorité sont renvoyées dans l'ordre de leur insertion. Puisque deux tâches ne peuvent jamais avoir le même numéro d'ajout, la comparaison des triplets ne va jamais chercher à comparer des tâches entre elles.

Une autre solution au fait que les tâches ne possèdent pas de relation d'ordre est de créer une classe d'encapsulation qui ignore l'élément tâche et ne compare que le champ priorité :

```
from dataclasses import dataclass, field
from typing import Any

@dataclass(order=True)
class PrioritizedItem:
    priority: int
    item: Any=field(compare=False)
```

Le problème restant consiste à trouver une tâche en attente et modifier sa priorité ou la supprimer. Trouver une tâche peut être réalisé à l'aide d'un dictionnaire pointant vers une entrée dans la file.

Supprimer une entrée ou changer sa priorité est plus difficile puisque cela romprait l'invariant de la structure de tas. Une solution possible est de marquer l'entrée comme supprimée et d'ajouter une nouvelle entrée avec sa priorité modifiée :

```
pq = []                         # list of entries arranged in a heap
entry_finder = {}               # mapping of tasks to entries
REMOVED = '<removed-task>'      # placeholder for a removed task
counter = itertools.count()     # unique sequence count

def add_task(task, priority=0):
    'Add a new task or update the priority of an existing task'
    if task in entry_finder:
        remove_task(task)
    count = next(counter)
    entry = [priority, count, task]
    entry_finder[task] = entry
    heappush(pq, entry)

def remove_task(task):
    'Mark an existing task as REMOVED.  Raise KeyError if not found.'
    entry = entry_finder.pop(task)
    entry[-1] = REMOVED

def pop_task():
    'Remove and return the lowest priority task. Raise KeyError if empty.'
    while pq:
        priority, count, task = heappop(pq)
        if task is not REMOVED:
            del entry_finder[task]
            return task
    raise KeyError('pop from an empty priority queue')
```

8.6.3 Théorie

Les tas sont des tableaux pour lesquels `a[k] <= a[2*k+1]` et `a[k] <= a[2*k+2]` pour tout *k* en comptant les éléments à partir de 0. Pour simplifier la comparaison, les éléments inexistants sont considérés comme étant infinis. L'intérêt des tas est que `a[0]` est toujours leur plus petit élément.

L'invariant étrange ci-dessus est une représentation efficace en mémoire d'un tournoi. Les nombres ci-dessous sont *k* et non `a[k]` :

```
                                0

              1                               2

      3               4                 5               6

  7       8       9       10      11      12      13      14

15 16   17 18   19 20   21 22   23 24   25 26   27 28   29 30
```

Dans l'arbre ci-dessus, chaque nœud *k* a pour enfants `2*k+1` et `2*k+2`. Dans les tournois binaires habituels dans les compétitions sportives, chaque nœud est le vainqueur des deux nœuds inférieurs et nous pouvons tracer le chemin du vainqueur le long de l'arbre afin de voir qui étaient ses adversaires. Cependant, dans de nombreuses applications informatiques de ces tournois, nous n'avons pas besoin de produire l'historique du vainqueur. Afin d'occuper moins de mémoire, on remplace le vainqueur lors de sa promotion par un autre élément à un plus bas niveau. La règle devient alors qu'un nœud et les deux nœuds qu'il chapeaute contiennent trois éléments différents, mais le nœud supérieur « gagne » contre les deux nœuds inférieurs.

If this heap invariant is protected at all time, index 0 is clearly the overall winner. The simplest algorithmic way to remove it and find the "next" winner is to move some loser (let's say cell 30 in the diagram above) into the 0 position, and then percolate this new 0 down the tree, exchanging values, until the invariant is re-established. This is clearly logarithmic on the total number of items in the tree. By iterating over all items, you get an $O(n \log n)$ sort.

Une propriété agréable de cet algorithme est qu'il est possible d'insérer efficacement de nouveaux éléments en cours de classement, du moment que les éléments insérés ne sont pas « meilleurs » que le dernier élément qui a été extrait. Ceci s'avère très utile dans des simulations où l'arbre contient la liste des événements arrivants et que la condition de « victoire » est le plus petit temps d'exécution planifié. Lorsqu'un événement programme l'exécution d'autres événements, ceux-ci sont planifiés pour le futur et peuvent donc rejoindre le tas. Ainsi, le tas est une bonne structure pour implémenter un ordonnanceur (et c'est ce que j'ai utilisé pour mon séquenceur MIDI ☺).

Plusieurs structures ont été étudiées en détail pour implémenter des ordonnanceurs et les tas sont bien adaptés : ils sont raisonnablement rapides, leur vitesse est presque constante et le pire cas ne diffère pas trop du cas moyen. S'il existe des représentations qui sont plus efficaces en général, les pires cas peuvent être terriblement mauvais.

Les tas sont également très utiles pour ordonner les données sur de gros disques. Vous savez probablement qu'un gros tri implique la production de séquences pré-classées (dont la taille est généralement liée à la quantité de mémoire CPU disponible), suivie par une passe de fusion qui est généralement organisée de façon très intelligente [1]. Il est très important que le classement initial produise des séquences les plus longues possibles. Les tournois sont une bonne façon d'arriver à ce résultat. Si, en utilisant toute la mémoire disponible pour stocker un tournoi, vous remplacez et faites percoler les éléments qui s'avèrent acceptables pour la séquence courante, vous produirez des séquences d'une taille égale au double de la mémoire pour une entrée aléatoire et bien mieux pour une entrée approximativement triée.

Qui plus est, si vous écrivez l'élément 0 sur le disque et que vous recevez en entrée un élément qui n'est pas adapté au tournoi actuel (parce que sa valeur « gagne » par rapport à la dernière valeur de sortie), alors il ne peut pas être stocké dans le tas donc la taille de ce dernier diminue. La mémoire libérée peut être réutilisée immédiatement pour progressivement construire un deuxième tas, qui croit à la même vitesse que le premier décroît. Lorsque le premier tas a complètement disparu, vous échangez les tas et démarrez une nouvelle séquence. Malin et plutôt efficace !

Pour résumer, les tas sont des structures de données qu'il est bon de connaître. Je les utilise dans quelques applications et je pense qu'il est bon de garder le module *heap* sous le coude. ☺

1. Les algorithmes de répartition de charge pour les disques, courants de nos jours, sont plus embêtants qu'utiles, en raison de la capacité des disques à réaliser des accès aléatoires. Sur les périphériques qui ne peuvent faire que de la lecture séquentielle, comme les gros lecteurs à bandes, le besoin était différent et il fallait être malin pour s'assurer (bien à l'avance) que chaque mouvement de bande serait le plus efficace possible (c'est-à-dire participerait au mieux à l'« avancée » de la fusion). Certaines cassettes pouvaient même lire à l'envers et cela était aussi utilisé pour éviter de remonter dans le temps. Croyez-moi, les bons tris sur bandes étaient spectaculaires à regarder ! Depuis la nuit des temps, trier a toujours été le Grand Art ! ☺

Notes

8.7 `bisect` --- Array bisection algorithm

Code source : Lib/bisect.py

This module provides support for maintaining a list in sorted order without having to sort the list after each insertion. For long lists of items with expensive comparison operations, this can be an improvement over linear searches or frequent resorting.

The module is called `bisect` because it uses a basic bisection algorithm to do its work. Unlike other bisection tools that search for a specific value, the functions in this module are designed to locate an insertion point. Accordingly, the functions never call an `__eq__()` method to determine whether a value has been found. Instead, the functions only call the `__lt__()` method and will return an insertion point between values in an array.

Les fonctions suivantes sont fournies :

`bisect.`**`bisect_left`**(*a*, *x*, *lo=0*, *hi=len(a)*, *, *key=None*)

Trouve le point d'insertion de *x* dans *a* permettant de conserver l'ordre. Les paramètres *lo* et *hi* permettent de limiter les emplacements à vérifier dans la liste, par défaut toute la liste est utilisée. Si *x* est déjà présent dans *a*, le point d'insertion proposé sera avant (à gauche) de l'entrée existante. Si *a* est déjà triée, la valeur renvoyée peut directement être utilisée comme premier paramètre de `list.insert()`.

The returned insertion point *ip* partitions the array *a* into two slices such that `all(elem < x for elem in a[lo : ip])` is true for the left slice and `all(elem >= x for elem in a[ip : hi])` is true for the right slice.

key specifies a *key function* of one argument that is used to extract a comparison key from each element in the array. To support searching complex records, the key function is not applied to the *x* value.

If *key* is `None`, the elements are compared directly and no key function is called.

Modifié dans la version 3.10 : ajout du paramètre *key*.

`bisect.`**`bisect_right`**(*a*, *x*, *lo=0*, *hi=len(a)*, *, *key=None*)

`bisect.`**`bisect`**(*a*, *x*, *lo=0*, *hi=len(a)*, *, *key=None*)

Similar to `bisect_left()`, but returns an insertion point which comes after (to the right of) any existing entries of *x* in *a*.

The returned insertion point *ip* partitions the array *a* into two slices such that `all(elem <= x for elem in a[lo : ip])` is true for the left slice and `all(elem > x for elem in a[ip : hi])` is true for the right slice.

Modifié dans la version 3.10 : ajout du paramètre *key*.

`bisect.`**`insort_left`**(*a*, *x*, *lo=0*, *hi=len(a)*, *, *key=None*)

Insère *x* dans *a* en préservant l'ordre.

This function first runs `bisect_left()` to locate an insertion point. Next, it runs the `insert()` method on *a* to insert *x* at the appropriate position to maintain sort order.

To support inserting records in a table, the *key* function (if any) is applied to *x* for the search step but not for the insertion step.

Keep in mind that the $O(\log n)$ search is dominated by the slow $O(n)$ insertion step.

Modifié dans la version 3.10 : ajout du paramètre *key*.

`bisect.`**`insort_right`**(*a*, *x*, *lo=0*, *hi=len(a)*, *, *key=None*)

`bisect.`**`insort`**(*a*, *x*, *lo=0*, *hi=len(a)*, *, *key=None*)

Similar to `insort_left()`, but inserting *x* in *a* after any existing entries of *x*.

This function first runs `bisect_right()` to locate an insertion point. Next, it runs the `insert()` method on *a* to insert *x* at the appropriate position to maintain sort order.

To support inserting records in a table, the *key* function (if any) is applied to *x* for the search step but not for the insertion step.

Keep in mind that the $O(\log n)$ search is dominated by the slow $O(n)$ insertion step.

Modifié dans la version 3.10 : ajout du paramètre *key*.

8.7.1 Notes sur la performance

Pour écrire du code sensible à la performance utilisant `bisect()` et `insort()`, prenez en compte ces quelques considérations :

— La bissection est une bonne idée pour rechercher une plage de valeurs. Pour une seule valeur, mieux vaut un dictionnaire.

— The *insort()* functions are $O(n)$ because the logarithmic search step is dominated by the linear time insertion step.

— The search functions are stateless and discard key function results after they are used. Consequently, if the search functions are used in a loop, the key function may be called again and again on the same array elements. If the key function isn't fast, consider wrapping it with `functools.cache()` to avoid duplicate computations. Alternatively, consider searching an array of precomputed keys to locate the insertion point (as shown in the examples section below).

Voir aussi

— Sorted Collections is a high performance module that uses *bisect* to managed sorted collections of data.

— SortedCollection recipe utilise le module *bisect* pour construire une classe collection exposant des méthodes de recherches naturelles et gérant une fonction clef. Les clefs sont pré-calculées pour économiser des appels inutiles à la fonction clef durant les recherches.

8.7.2 Chercher dans des listes triées

The above *bisect functions* are useful for finding insertion points but can be tricky or awkward to use for common searching tasks. The following five functions show how to transform them into the standard lookups for sorted lists :

```
def index(a, x):
    'Locate the leftmost value exactly equal to x'
    i = bisect_left(a, x)
    if i != len(a) and a[i] == x:
        return i
    raise ValueError

def find_lt(a, x):
    'Find rightmost value less than x'
    i = bisect_left(a, x)
    if i:
        return a[i-1]
    raise ValueError

def find_le(a, x):
    'Find rightmost value less than or equal to x'
    i = bisect_right(a, x)
    if i:
        return a[i-1]
    raise ValueError

def find_gt(a, x):
    'Find leftmost value greater than x'
    i = bisect_right(a, x)
    if i != len(a):
```

(suite sur la page suivante)

(suite de la page précédente)

```
        return a[i]
    raise ValueError

def find_ge(a, x):
    'Find leftmost item greater than or equal to x'
    i = bisect_left(a, x)
    if i != len(a):
        return a[i]
    raise ValueError
```

8.7.3 Exemples

The `bisect()` function can be useful for numeric table lookups. This example uses `bisect()` to look up a letter grade for an exam score (say) based on a set of ordered numeric breakpoints : 90 and up is an 'A', 80 to 89 is a 'B', and so on :

```
>>> def grade(score, breakpoints=[60, 70, 80, 90], grades='FDCBA'):
...     i = bisect(breakpoints, score)
...     return grades[i]
...
>>> [grade(score) for score in [33, 99, 77, 70, 89, 90, 100]]
['F', 'A', 'C', 'C', 'B', 'A', 'A']
```

The `bisect()` and `insort()` functions also work with lists of tuples. The *key* argument can serve to extract the field used for ordering records in a table :

```
>>> from collections import namedtuple
>>> from operator import attrgetter
>>> from bisect import bisect, insort
>>> from pprint import pprint

>>> Movie = namedtuple('Movie', ('name', 'released', 'director'))

>>> movies = [
...     Movie('Jaws', 1975, 'Spielberg'),
...     Movie('Titanic', 1997, 'Cameron'),
...     Movie('The Birds', 1963, 'Hitchcock'),
...     Movie('Aliens', 1986, 'Cameron')
... ]

>>> # Find the first movie released after 1960
>>> by_year = attrgetter('released')
>>> movies.sort(key=by_year)
>>> movies[bisect(movies, 1960, key=by_year)]
Movie(name='The Birds', released=1963, director='Hitchcock')

>>> # Insert a movie while maintaining sort order
>>> romance = Movie('Love Story', 1970, 'Hiller')
>>> insort(movies, romance, key=by_year)
>>> pprint(movies)
[Movie(name='The Birds', released=1963, director='Hitchcock'),
 Movie(name='Love Story', released=1970, director='Hiller'),
 Movie(name='Jaws', released=1975, director='Spielberg'),
 Movie(name='Aliens', released=1986, director='Cameron'),
 Movie(name='Titanic', released=1997, director='Cameron')]
```

If the key function is expensive, it is possible to avoid repeated function calls by searching a list of precomputed keys

to find the index of a record :

```
>>> data = [('red', 5), ('blue', 1), ('yellow', 8), ('black', 0)]
>>> data.sort(key=lambda r: r[1])       # Or use operator.itemgetter(1).
>>> keys = [r[1] for r in data]         # Precompute a list of keys.
>>> data[bisect_left(keys, 0)]
('black', 0)
>>> data[bisect_left(keys, 1)]
('blue', 1)
>>> data[bisect_left(keys, 5)]
('red', 5)
>>> data[bisect_left(keys, 8)]
('yellow', 8)
```

8.8 array --- Efficient arrays of numeric values

This module defines an object type which can compactly represent an array of basic values : characters, integers, floating-point numbers. Arrays are sequence types and behave very much like lists, except that the type of objects stored in them is constrained. The type is specified at object creation time by using a *type code*, which is a single character. The following type codes are defined :

Code d'indication du type	Type C	Type Python	Taille minimum en octets	Notes
'b'	signed char	*int*	1	
'B'	unsigned char	*int*	1	
'u'	wchar_t	Caractère Unicode	2	(1)
'w'	Py_UCS4	Caractère Unicode	4	
'h'	signed short	*int*	2	
'H'	unsigned short	*int*	2	
'i'	signed int	*int*	2	
'I'	unsigned int	*int*	2	
'l'	signed long	*int*	4	
'L'	unsigned long	*int*	4	
'q'	signed long long	*int*	8	
'Q'	unsigned long long	*int*	8	
'f'	*float*	*float*	4	
'd'	double	*float*	8	

Notes :

(1) It can be 16 bits or 32 bits depending on the platform.

Modifié dans la version 3.9 : `array('u')` now uses `wchar_t` as C type instead of deprecated `Py_UNICODE`. This change doesn't affect its behavior because `Py_UNICODE` is alias of `wchar_t` since Python 3.3.

Deprecated since version 3.3, will be removed in version 3.16 : Please migrate to `'w'` typecode.

The actual representation of values is determined by the machine architecture (strictly speaking, by the C implementation). The actual size can be accessed through the *array.itemsize* attribute.

The module defines the following item :

`array.typecodes`

Une chaîne avec tous les codes de types disponibles.

Le module définit le type suivant :

class array.**array**(*typecode*[, *initializer*])

A new array whose items are restricted by *typecode*, and initialized from the optional *initializer* value, which must be a `bytes` or `bytearray` object, a Unicode string, or iterable over elements of the appropriate type.

If given a `bytes` or `bytearray` object, the initializer is passed to the new array's `frombytes()` method ; if given a Unicode string, the initializer is passed to the `fromunicode()` method ; otherwise, the initializer's iterator is passed to the `extend()` method to add initial items to the array.

Les objets de tableau supportent les opérations classiques de séquence : indiçage, découpage, concaténation et multiplication. Lors de l'utilisation de tranche, la valeur assignée doit être un tableau du même type ; dans tous les autres cas, l'exception `TypeError` est levée. Les objets de tableau implémentent également l'interface tampon, et peuvent être utilisés partout où *bytes-like objects* sont supportés.

Lève un *événement d'audit* `array.__new__` avec les arguments `typecode`, `initializer`.

typecode

Le code (de type Python caractère) utilisé pour spécifier le type des éléments du tableau.

itemsize

La longueur en octets d'un élément du tableau dans la représentation interne.

append(*x*)

Ajoute un nouvel élément avec la valeur *x* à la fin du tableau.

buffer_info()

Return a tuple `(address, length)` giving the current memory address and the length in elements of the buffer used to hold array's contents. The size of the memory buffer in bytes can be computed as `array.buffer_info()[1] * array.itemsize`. This is occasionally useful when working with low-level (and inherently unsafe) I/O interfaces that require memory addresses, such as certain `ioctl()` operations. The returned numbers are valid as long as the array exists and no length-changing operations are applied to it.

> **Note**
>
> Lors de l'utilisation d'objets tableaux provenant de codes écrits en C ou C++ (le seul moyen d'utiliser efficacement ces informations), il est plus logique d'utiliser l'interface tampon supportée par les objets tableaux. Cette méthode est maintenue pour des raisons de rétrocompatibilité et devrait être évitée dans un nouveau code. L'interface tampon est documentée dans bufferobjects.

byteswap()

Boutisme de tous les éléments du tableau. Ceci n'est pris en charge que pour les valeurs de 1, 2, 4 ou 8 octets ; pour les autres types de valeur, `RuntimeError` est levée. Il est utile lors de la lecture de données à partir d'un fichier écrit sur une machine avec un ordre d'octets différent.

count(*x*)

Renvoi le nombre d'occurrences de *x* dans le tableau.

extend(*iterable*)

Ajoute les éléments de *iterable* à la fin du tableau. Si *iterable* est un autre tableau, il doit le même code d'indication du type ; dans le cas contraire, `TypeError` sera levée. Si *iterable* n'est pas un tableau, il doit être itérable et ces éléments doivent être du bon type pour être ajoutés dans le tableau.

frombytes(*buffer*)

Appends items from the *bytes-like object*, interpreting its content as an array of machine values (as if it had been read from a file using the `fromfile()` method).

Ajouté dans la version 3.2 : `fromstring()` is renamed to `frombytes()` for clarity.

fromfile(*f*, *n*)

Read *n* items (as machine values) from the *file object* *f* and append them to the end of the array. If less

than *n* items are available, `EOFError` is raised, but the items that were available are still inserted into the array.

fromlist(*list*)

Ajoute les éléments de la liste. C'est l'équivalent de `for x in list: a.append(x)` sauf que s'il y a une erreur de type, le tableau est inchangé.

fromunicode(*s*)

Extends this array with data from the given Unicode string. The array must have type code `'u'` or `'w'`; otherwise a `ValueError` is raised. Use `array.frombytes(unicodestring.encode(enc))` to append Unicode data to an array of some other type.

index(*x*[, *start*[, *stop*]])

Return the smallest *i* such that *i* is the index of the first occurrence of *x* in the array. The optional arguments *start* and *stop* can be specified to search for *x* within a subsection of the array. Raise `ValueError` if *x* is not found.

Modifié dans la version 3.10 : Added optional *start* and *stop* parameters.

insert(*i*, *x*)

Ajoute un nouvel élément avec la valeur *x* dans le tableau avant la position *i*. Les valeurs négatives sont traitées relativement à la fin du tableau.

pop([*i*])

Supprime l'élément du tableau avec l'index *i* et le renvoie. L'argument optionnel par défaut est à `-1`, de sorte que par défaut le dernier élément est supprimé et renvoyé.

remove(*x*)

Supprime la première occurrence de *x* du tableau.

clear()

Remove all elements from the array.

Ajouté dans la version 3.13.

reverse()

Inverse l'ordre des éléments du tableau.

tobytes()

Convertit le tableau en un tableau de valeurs machine et renvoie la représentation en octets (la même séquence d'octets qui serait écrite par la méthode `tofile()`).

Ajouté dans la version 3.2 : `tostring()` is renamed to `tobytes()` for clarity.

tofile(*f*)

Écrit tous les éléments (en tant que valeurs machine) du *file object* *f*.

tolist()

Convertit le tableau en une liste ordinaire avec les mêmes éléments.

tounicode()

Convert the array to a Unicode string. The array must have a type `'u'` or `'w'`; otherwise a `ValueError` is raised. Use `array.tobytes().decode(enc)` to obtain a Unicode string from an array of some other type.

The string representation of array objects has the form `array(typecode, initializer)`. The *initializer* is omitted if the array is empty, otherwise it is a Unicode string if the *typecode* is `'u'` or `'w'`, otherwise it is a list of numbers. The string representation is guaranteed to be able to be converted back to an array with the same type and value using `eval()`, so long as the `array` class has been imported using `from array import array`. Variables `inf` and `nan` must also be defined if it contains corresponding floating-point values. Examples :

```
array('l')
array('w', 'hello \u2641')
```

(suite sur la page suivante)

(suite de la page précédente)

```
array('l', [1, 2, 3, 4, 5])
array('d', [1.0, 2.0, 3.14, -inf, nan])
```

Voir aussi

Module *struct*
Empaquetage et dépaquetage de données binaires hétérogènes.

NumPy
The NumPy package defines another array type.

8.9 weakref --- Weak references

Code source : Lib/weakref.py

The *weakref* module allows the Python programmer to create *weak references* to objects.

In the following, the term *referent* means the object which is referred to by a weak reference.

A weak reference to an object is not enough to keep the object alive : when the only remaining references to a referent are weak references, *garbage collection* is free to destroy the referent and reuse its memory for something else. However, until the object is actually destroyed the weak reference may return the object even if there are no strong references to it.

A primary use for weak references is to implement caches or mappings holding large objects, where it's desired that a large object not be kept alive solely because it appears in a cache or mapping.

For example, if you have a number of large binary image objects, you may wish to associate a name with each. If you used a Python dictionary to map names to images, or images to names, the image objects would remain alive just because they appeared as values or keys in the dictionaries. The *WeakKeyDictionary* and *WeakValueDictionary* classes supplied by the *weakref* module are an alternative, using weak references to construct mappings that don't keep objects alive solely because they appear in the mapping objects. If, for example, an image object is a value in a *WeakValueDictionary*, then when the last remaining references to that image object are the weak references held by weak mappings, garbage collection can reclaim the object, and its corresponding entries in weak mappings are simply deleted.

WeakKeyDictionary and *WeakValueDictionary* use weak references in their implementation, setting up callback functions on the weak references that notify the weak dictionaries when a key or value has been reclaimed by garbage collection. *WeakSet* implements the *set* interface, but keeps weak references to its elements, just like a *WeakKeyDictionary* does.

finalize provides a straight forward way to register a cleanup function to be called when an object is garbage collected. This is simpler to use than setting up a callback function on a raw weak reference, since the module automatically ensures that the finalizer remains alive until the object is collected.

Most programs should find that using one of these weak container types or *finalize* is all they need -- it's not usually necessary to create your own weak references directly. The low-level machinery is exposed by the *weakref* module for the benefit of advanced uses.

Not all objects can be weakly referenced. Objects which support weak references include class instances, functions written in Python (but not in C), instance methods, sets, frozensets, some *file objects*, *generators*, type objects, sockets, arrays, deques, regular expression pattern objects, and code objects.

Modifié dans la version 3.2 : Added support for thread.lock, threading.Lock, and code objects.

Several built-in types such as *list* and *dict* do not directly support weak references but can add support through subclassing :

```
class Dict(dict):
    pass

obj = Dict(red=1, green=2, blue=3)   # this object is weak referenceable
```

Particularité de l'implémentation CPython : Other built-in types such as `tuple` and `int` do not support weak references even when subclassed.

Extension types can easily be made to support weak references ; see weakref-support.

When `__slots__` are defined for a given type, weak reference support is disabled unless a `'__weakref__'` string is also present in the sequence of strings in the `__slots__` declaration. See __slots__ documentation for details.

class `weakref.`**`ref`**(*object*[, *callback*])

Return a weak reference to *object*. The original object can be retrieved by calling the reference object if the referent is still alive ; if the referent is no longer alive, calling the reference object will cause `None` to be returned. If *callback* is provided and not `None`, and the returned weakref object is still alive, the callback will be called when the object is about to be finalized ; the weak reference object will be passed as the only parameter to the callback ; the referent will no longer be available.

It is allowable for many weak references to be constructed for the same object. Callbacks registered for each weak reference will be called from the most recently registered callback to the oldest registered callback.

Exceptions raised by the callback will be noted on the standard error output, but cannot be propagated ; they are handled in exactly the same way as exceptions raised from an object's `__del__()` method.

Weak references are *hashable* if the *object* is hashable. They will maintain their hash value even after the *object* was deleted. If `hash()` is called the first time only after the *object* was deleted, the call will raise `TypeError`.

Weak references support tests for equality, but not ordering. If the referents are still alive, two references have the same equality relationship as their referents (regardless of the *callback*). If either referent has been deleted, the references are equal only if the reference objects are the same object.

This is a subclassable type rather than a factory function.

`__callback__`

This read-only attribute returns the callback currently associated to the weakref. If there is no callback or if the referent of the weakref is no longer alive then this attribute will have value `None`.

Modifié dans la version 3.4 : Added the `__callback__` attribute.

`weakref.`**`proxy`**(*object*[, *callback*])

Return a proxy to *object* which uses a weak reference. This supports use of the proxy in most contexts instead of requiring the explicit dereferencing used with weak reference objects. The returned object will have a type of either `ProxyType` or `CallableProxyType`, depending on whether *object* is callable. Proxy objects are not *hashable* regardless of the referent ; this avoids a number of problems related to their fundamentally mutable nature, and prevents their use as dictionary keys. *callback* is the same as the parameter of the same name to the `ref()` function.

Accessing an attribute of the proxy object after the referent is garbage collected raises `ReferenceError`.

Modifié dans la version 3.8 : Extended the operator support on proxy objects to include the matrix multiplication operators `@` and `@=`.

`weakref.`**`getweakrefcount`**(*object*)

Return the number of weak references and proxies which refer to *object*.

`weakref.`**`getweakrefs`**(*object*)

Return a list of all weak reference and proxy objects which refer to *object*.

class `weakref.`**`WeakKeyDictionary`**([*dict*])

Mapping class that references keys weakly. Entries in the dictionary will be discarded when there is no longer a strong reference to the key. This can be used to associate additional data with an object owned by other parts

of an application without adding attributes to those objects. This can be especially useful with objects that override attribute accesses.

Note that when a key with equal value to an existing key (but not equal identity) is inserted into the dictionary, it replaces the value but does not replace the existing key. Due to this, when the reference to the original key is deleted, it also deletes the entry in the dictionary :

```
>>> class T(str): pass
...
>>> k1, k2 = T(), T()
>>> d = weakref.WeakKeyDictionary()
>>> d[k1] = 1   # d = {k1: 1}
>>> d[k2] = 2   # d = {k1: 2}
>>> del k1      # d = {}
```

A workaround would be to remove the key prior to reassignment :

```
>>> class T(str): pass
...
>>> k1, k2 = T(), T()
>>> d = weakref.WeakKeyDictionary()
>>> d[k1] = 1   # d = {k1: 1}
>>> del d[k1]
>>> d[k2] = 2   # d = {k2: 2}
>>> del k1      # d = {k2: 2}
```

Modifié dans la version 3.9 : Added support for `|` and `|=` operators, as specified in **PEP 584**.

`WeakKeyDictionary` objects have an additional method that exposes the internal references directly. The references are not guaranteed to be "live" at the time they are used, so the result of calling the references needs to be checked before being used. This can be used to avoid creating references that will cause the garbage collector to keep the keys around longer than needed.

`WeakKeyDictionary.`**`keyrefs`**`()`

Return an iterable of the weak references to the keys.

`class` `weakref.`**`WeakValueDictionary`**`(`[*dict*]`)`

Mapping class that references values weakly. Entries in the dictionary will be discarded when no strong reference to the value exists any more.

Modifié dans la version 3.9 : Added support for `|` and `|=` operators, as specified in **PEP 584**.

`WeakValueDictionary` objects have an additional method that has the same issues as the `WeakKeyDictionary.keyrefs()` method.

`WeakValueDictionary.`**`valuerefs`**`()`

Return an iterable of the weak references to the values.

`class` `weakref.`**`WeakSet`**`(`[*elements*]`)`

Set class that keeps weak references to its elements. An element will be discarded when no strong reference to it exists any more.

`class` `weakref.`**`WeakMethod`**`(`*method*[, *callback*]`)`

A custom `ref` subclass which simulates a weak reference to a bound method (i.e., a method defined on a class and looked up on an instance). Since a bound method is ephemeral, a standard weak reference cannot keep hold of it. `WeakMethod` has special code to recreate the bound method until either the object or the original function dies :

```
>>> class C:
...     def method(self):
...         print("method called!")
```

(suite sur la page suivante)

(suite de la page précédente)

```
...
>>> c = C()
>>> r = weakref.ref(c.method)
>>> r()
>>> r = weakref.WeakMethod(c.method)
>>> r()
<bound method C.method of <__main__.C object at 0x7fc859830220>>
>>> r()()
method called!
>>> del c
>>> gc.collect()
0
>>> r()
>>>
```

callback is the same as the parameter of the same name to the `ref()` function.

Ajouté dans la version 3.4.

class `weakref.`**`finalize`**(*obj*, *func*, /, **args*, ***kwargs*)

Return a callable finalizer object which will be called when *obj* is garbage collected. Unlike an ordinary weak reference, a finalizer will always survive until the reference object is collected, greatly simplifying lifecycle management.

A finalizer is considered *alive* until it is called (either explicitly or at garbage collection), and after that it is *dead*. Calling a live finalizer returns the result of evaluating `func(*arg, **kwargs)`, whereas calling a dead finalizer returns `None`.

Exceptions raised by finalizer callbacks during garbage collection will be shown on the standard error output, but cannot be propagated. They are handled in the same way as exceptions raised from an object's `__del__()` method or a weak reference's callback.

When the program exits, each remaining live finalizer is called unless its `atexit` attribute has been set to false. They are called in reverse order of creation.

A finalizer will never invoke its callback during the later part of the *interpreter shutdown* when module globals are liable to have been replaced by `None`.

`__call__`()

If *self* is alive then mark it as dead and return the result of calling `func(*args, **kwargs)`. If *self* is dead then return `None`.

`detach`()

If *self* is alive then mark it as dead and return the tuple `(obj, func, args, kwargs)`. If *self* is dead then return `None`.

`peek`()

If *self* is alive then return the tuple `(obj, func, args, kwargs)`. If *self* is dead then return `None`.

`alive`

Property which is true if the finalizer is alive, false otherwise.

`atexit`

A writable boolean property which by default is true. When the program exits, it calls all remaining live finalizers for which `atexit` is true. They are called in reverse order of creation.

> **Note**
>
> It is important to ensure that *func*, *args* and *kwargs* do not own any references to *obj*, either directly or indirectly, since otherwise *obj* will never be garbage collected. In particular, *func* should not be a bound

method of *obj*.

Ajouté dans la version 3.4.

`weakref.ReferenceType`

The type object for weak references objects.

`weakref.ProxyType`

The type object for proxies of objects which are not callable.

`weakref.CallableProxyType`

The type object for proxies of callable objects.

`weakref.ProxyTypes`

Sequence containing all the type objects for proxies. This can make it simpler to test if an object is a proxy without being dependent on naming both proxy types.

Voir aussi

PEP 205 - Weak References
The proposal and rationale for this feature, including links to earlier implementations and information about similar features in other languages.

8.9.1 Objets à références faibles

Weak reference objects have no methods and no attributes besides `ref.__callback__`. A weak reference object allows the referent to be obtained, if it still exists, by calling it :

```
>>> import weakref
>>> class Object:
...     pass
...
>>> o = Object()
>>> r = weakref.ref(o)
>>> o2 = r()
>>> o is o2
True
```

If the referent no longer exists, calling the reference object returns `None` :

```
>>> del o, o2
>>> print(r())
None
```

Testing that a weak reference object is still live should be done using the expression `ref() is not None`. Normally, application code that needs to use a reference object should follow this pattern :

```
# r is a weak reference object
o = r()
if o is None:
    # referent has been garbage collected
    print("Object has been deallocated; can't frobnicate.")
else:
    print("Object is still live!")
    o.do_something_useful()
```

Using a separate test for "liveness" creates race conditions in threaded applications ; another thread can cause a weak reference to become invalidated before the weak reference is called ; the idiom shown above is safe in threaded applications as well as single-threaded applications.

Specialized versions of `ref` objects can be created through subclassing. This is used in the implementation of the `WeakValueDictionary` to reduce the memory overhead for each entry in the mapping. This may be most useful to associate additional information with a reference, but could also be used to insert additional processing on calls to retrieve the referent.

This example shows how a subclass of `ref` can be used to store additional information about an object and affect the value that's returned when the referent is accessed :

```
import weakref

class ExtendedRef(weakref.ref):
    def __init__(self, ob, callback=None, /, **annotations):
        super().__init__(ob, callback)
        self.__counter = 0
        for k, v in annotations.items():
            setattr(self, k, v)

    def __call__(self):
        """Return a pair containing the referent and the number of
        times the reference has been called.
        """
        ob = super().__call__()
        if ob is not None:
            self.__counter += 1
            ob = (ob, self.__counter)
        return ob
```

8.9.2 Exemple

This simple example shows how an application can use object IDs to retrieve objects that it has seen before. The IDs of the objects can then be used in other data structures without forcing the objects to remain alive, but the objects can still be retrieved by ID if they do.

```
import weakref

_id2obj_dict = weakref.WeakValueDictionary()

def remember(obj):
    oid = id(obj)
    _id2obj_dict[oid] = obj
    return oid

def id2obj(oid):
    return _id2obj_dict[oid]
```

8.9.3 Finalizer Objects

The main benefit of using `finalize` is that it makes it simple to register a callback without needing to preserve the returned finalizer object. For instance

```
>>> import weakref
>>> class Object:
...     pass
...
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> kenny = Object()
>>> weakref.finalize(kenny, print, "You killed Kenny!")
<finalize object at ...; for 'Object' at ...>
>>> del kenny
You killed Kenny!
```

The finalizer can be called directly as well. However the finalizer will invoke the callback at most once.

```
>>> def callback(x, y, z):
...     print("CALLBACK")
...     return x + y + z
...
>>> obj = Object()
>>> f = weakref.finalize(obj, callback, 1, 2, z=3)
>>> assert f.alive
>>> assert f() == 6
CALLBACK
>>> assert not f.alive
>>> f()                     # callback not called because finalizer dead
>>> del obj                 # callback not called because finalizer dead
```

You can unregister a finalizer using its `detach()` method. This kills the finalizer and returns the arguments passed to the constructor when it was created.

```
>>> obj = Object()
>>> f = weakref.finalize(obj, callback, 1, 2, z=3)
>>> f.detach()
(<...Object object ...>, <function callback ...>, (1, 2), {'z': 3})
>>> newobj, func, args, kwargs = _
>>> assert not f.alive
>>> assert newobj is obj
>>> assert func(*args, **kwargs) == 6
CALLBACK
```

Unless you set the `atexit` attribute to `False`, a finalizer will be called when the program exits if it is still alive. For instance

```
>>> obj = Object()
>>> weakref.finalize(obj, print, "obj dead or exiting")
<finalize object at ...; for 'Object' at ...>
>>> exit()
obj dead or exiting
```

8.9.4 Comparing finalizers with `__del__()` methods

Suppose we want to create a class whose instances represent temporary directories. The directories should be deleted with their contents when the first of the following events occurs :

— the object is garbage collected,
— the object's `remove()` method is called, or
— the program exits.

We might try to implement the class using a `__del__()` method as follows :

```
class TempDir:
    def __init__(self):
        self.name = tempfile.mkdtemp()

    def remove(self):
```

(suite sur la page suivante)

(suite de la page précédente)

```
        if self.name is not None:
            shutil.rmtree(self.name)
            self.name = None

    @property
    def removed(self):
        return self.name is None

    def __del__(self):
        self.remove()
```

Starting with Python 3.4, `__del__()` methods no longer prevent reference cycles from being garbage collected, and module globals are no longer forced to `None` during *interpreter shutdown*. So this code should work without any issues on CPython.

However, handling of `__del__()` methods is notoriously implementation specific, since it depends on internal details of the interpreter's garbage collector implementation.

A more robust alternative can be to define a finalizer which only references the specific functions and objects that it needs, rather than having access to the full state of the object :

```
class TempDir:
    def __init__(self):
        self.name = tempfile.mkdtemp()
        self._finalizer = weakref.finalize(self, shutil.rmtree, self.name)

    def remove(self):
        self._finalizer()

    @property
    def removed(self):
        return not self._finalizer.alive
```

Defined like this, our finalizer only receives a reference to the details it needs to clean up the directory appropriately. If the object never gets garbage collected the finalizer will still be called at exit.

The other advantage of weakref based finalizers is that they can be used to register finalizers for classes where the definition is controlled by a third party, such as running code when a module is unloaded :

```
import weakref, sys
def unloading_module():
    # implicit reference to the module globals from the function body
weakref.finalize(sys.modules[__name__], unloading_module)
```

Note

If you create a finalizer object in a daemonic thread just as the program exits then there is the possibility that the finalizer does not get called at exit. However, in a daemonic thread `atexit.register()`, `try: ... finally: ...` and `with: ...` do not guarantee that cleanup occurs either.

8.10 `types` --- Dynamic type creation and names for built-in types

Code source : Lib/types.py

This module defines utility functions to assist in dynamic creation of new types.

It also defines names for some object types that are used by the standard Python interpreter, but not exposed as builtins like `int` or `str` are.

Finally, it provides some additional type-related utility classes and functions that are not fundamental enough to be builtins.

8.10.1 Dynamic Type Creation

`types.new_class`(*name*, *bases=()*, *kwds=None*, *exec_body=None*)

Creates a class object dynamically using the appropriate metaclass.

The first three arguments are the components that make up a class definition header : the class name, the base classes (in order), the keyword arguments (such as `metaclass`).

The *exec_body* argument is a callback that is used to populate the freshly created class namespace. It should accept the class namespace as its sole argument and update the namespace directly with the class contents. If no callback is provided, it has the same effect as passing in `lambda ns: None`.

Ajouté dans la version 3.3.

`types.prepare_class`(*name*, *bases=()*, *kwds=None*)

Calculates the appropriate metaclass and creates the class namespace.

The arguments are the components that make up a class definition header : the class name, the base classes (in order) and the keyword arguments (such as `metaclass`).

The return value is a 3-tuple : `metaclass, namespace, kwds`

metaclass is the appropriate metaclass, *namespace* is the prepared class namespace and *kwds* is an updated copy of the passed in *kwds* argument with any `'metaclass'` entry removed. If no *kwds* argument is passed in, this will be an empty dict.

Ajouté dans la version 3.3.

Modifié dans la version 3.6 : The default value for the `namespace` element of the returned tuple has changed. Now an insertion-order-preserving mapping is used when the metaclass does not have a `__prepare__` method.

Voir aussi

metaclasses
Full details of the class creation process supported by these functions

PEP 3115 — Méta-classes dans Python 3000
introduction de la fonction automatique `__prepare__` de l'espace de nommage

`types.resolve_bases`(*bases*)

Resolve MRO entries dynamically as specified by **PEP 560**.

This function looks for items in *bases* that are not instances of `type`, and returns a tuple where each such object that has an `__mro_entries__()` method is replaced with an unpacked result of calling this method. If a *bases* item is an instance of `type`, or it doesn't have an `__mro_entries__()` method, then it is included in the return tuple unchanged.

Ajouté dans la version 3.7.

`types.get_original_bases`(*cls*, /)

Return the tuple of objects originally given as the bases of *cls* before the `__mro_entries__()` method has been called on any bases (following the mechanisms laid out in **PEP 560**). This is useful for introspecting *Generics*.

For classes that have an `__orig_bases__` attribute, this function returns the value of `cls.__orig_bases__`. For classes without the `__orig_bases__` attribute, `cls.__bases__` is returned.

Exemples :

```
from typing import TypeVar, Generic, NamedTuple, TypedDict

T = TypeVar("T")
class Foo(Generic[T]): ...
class Bar(Foo[int], float): ...
class Baz(list[str]): ...
Eggs = NamedTuple("Eggs", [("a", int), ("b", str)])
Spam = TypedDict("Spam", {"a": int, "b": str})

assert Bar.__bases__ == (Foo, float)
assert get_original_bases(Bar) == (Foo[int], float)

assert Baz.__bases__ == (list,)
assert get_original_bases(Baz) == (list[str],)

assert Eggs.__bases__ == (tuple,)
assert get_original_bases(Eggs) == (NamedTuple,)

assert Spam.__bases__ == (dict,)
assert get_original_bases(Spam) == (TypedDict,)

assert int.__bases__ == (object,)
assert get_original_bases(int) == (object,)
```

Ajouté dans la version 3.12.

Voir aussi

PEP 560 — Gestion de base pour les types modules et les types génériques

8.10.2 Standard Interpreter Types

This module provides names for many of the types that are required to implement a Python interpreter. It deliberately avoids including some of the types that arise only incidentally during processing such as the `listiterator` type.

Typical use of these names is for *`isinstance()`* or *`issubclass()`* checks.

If you instantiate any of these types, note that signatures may vary between Python versions.

Standard names are defined for the following types :

`types.`**`NoneType`**

The type of *`None`*.

Ajouté dans la version 3.10.

`types.`**`FunctionType`**

`types.`**`LambdaType`**

The type of user-defined functions and functions created by `lambda` expressions.

Raises an *auditing event* `function.__new__` with argument `code`.

The audit event only occurs for direct instantiation of function objects, and is not raised for normal compilation.

`types.`**`GeneratorType`**

The type of *generator*-iterator objects, created by generator functions.

`types.`**`CoroutineType`**

The type of *coroutine* objects, created by `async def` functions.

Ajouté dans la version 3.5.

types.**AsyncGeneratorType**

The type of *asynchronous generator*-iterator objects, created by asynchronous generator functions.

Ajouté dans la version 3.6.

class types.**CodeType**(***kwargs*)

The type of code objects such as returned by `compile()`.

Raises an *auditing event* `code.__new__` with arguments `code`, `filename`, `name`, `argcount`, `posonlyargcount`, `kwonlyargcount`, `nlocals`, `stacksize`, `flags`.

Note that the audited arguments may not match the names or positions required by the initializer. The audit event only occurs for direct instantiation of code objects, and is not raised for normal compilation.

types.**CellType**

The type for cell objects : such objects are used as containers for a function's *closure variables*.

Ajouté dans la version 3.8.

types.**MethodType**

The type of methods of user-defined class instances.

types.**BuiltinFunctionType**

types.**BuiltinMethodType**

The type of built-in functions like `len()` or `sys.exit()`, and methods of built-in classes. (Here, the term "built-in" means "written in C".)

types.**WrapperDescriptorType**

The type of methods of some built-in data types and base classes such as `object.__init__()` or `object.__lt__()`.

Ajouté dans la version 3.7.

types.**MethodWrapperType**

The type of *bound* methods of some built-in data types and base classes. For example it is the type of `object().__str__`.

Ajouté dans la version 3.7.

types.**NotImplementedType**

The type of `NotImplemented`.

Ajouté dans la version 3.10.

types.**MethodDescriptorType**

The type of methods of some built-in data types such as `str.join()`.

Ajouté dans la version 3.7.

types.**ClassMethodDescriptorType**

The type of *unbound* class methods of some built-in data types such as `dict.__dict__['fromkeys']`.

Ajouté dans la version 3.7.

class types.**ModuleType**(*name*, *doc=None*)

The type of *modules*. The constructor takes the name of the module to be created and optionally its *docstring*.

Voir aussi

Documentation on module objects
Provides details on the special attributes that can be found on instances of `ModuleType`.

> ***importlib.util.module_from_spec()***
> Modules created using the `ModuleType` constructor are created with many of their special attributes unset or set to default values. `module_from_spec()` provides a more robust way of creating `ModuleType` instances which ensures the various attributes are set appropriately.

`types.`**`EllipsisType`**

The type of *`Ellipsis`*.

Ajouté dans la version 3.10.

class `types.`**`GenericAlias`**(*t_origin*, *t_args*)

The type of *parameterized generics* such as `list[int]`.

`t_origin` should be a non-parameterized generic class, such as `list`, `tuple` or `dict`. `t_args` should be a *`tuple`* (possibly of length 1) of types which parameterize `t_origin` :

```
>>> from types import GenericAlias

>>> list[int] == GenericAlias(list, (int,))
True
>>> dict[str, int] == GenericAlias(dict, (str, int))
True
```

Ajouté dans la version 3.9.

Modifié dans la version 3.9.2 : This type can now be subclassed.

> **Voir aussi**
>
> ***Types alias génériques***
> In-depth documentation on instances of `types.GenericAlias`
>
> **PEP 585 – Types génériques d'indication de type dans les conteneurs standard (page en anglais)**
> Introducing the `types.GenericAlias` class

class `types.`**`UnionType`**

The type of *union type expressions*.

Ajouté dans la version 3.10.

class `types.`**`TracebackType`**(*tb_next*, *tb_frame*, *tb_lasti*, *tb_lineno*)

The type of traceback objects such as found in `sys.exception().__traceback__`.

See the language reference for details of the available attributes and operations, and guidance on creating tracebacks dynamically.

`types.`**`FrameType`**

The type of frame objects such as found in `tb.tb_frame` if `tb` is a traceback object.

`types.`**`GetSetDescriptorType`**

The type of objects defined in extension modules with `PyGetSetDef`, such as `FrameType.f_locals` or `array.array.typecode`. This type is used as descriptor for object attributes ; it has the same purpose as the *`property`* type, but for classes defined in extension modules.

`types.`**`MemberDescriptorType`**

The type of objects defined in extension modules with `PyMemberDef`, such as `datetime.timedelta.days`. This type is used as descriptor for simple C data members which use standard conversion functions ; it has the same purpose as the *`property`* type, but for classes defined in extension modules.

In addition, when a class is defined with a `__slots__` attribute, then for each slot, an instance of

`MemberDescriptorType` will be added as an attribute on the class. This allows the slot to appear in the class's `__dict__`.

Particularité de l'implémentation CPython : In other implementations of Python, this type may be identical to `GetSetDescriptorType`.

class `types.`**`MappingProxyType`**(*mapping*)

Read-only proxy of a mapping. It provides a dynamic view on the mapping's entries, which means that when the mapping changes, the view reflects these changes.

Ajouté dans la version 3.3.

Modifié dans la version 3.9 : Updated to support the new union (`|`) operator from **PEP 584**, which simply delegates to the underlying mapping.

key in proxy

Return `True` if the underlying mapping has a key *key*, else `False`.

proxy[key]

Return the item of the underlying mapping with key *key*. Raises a `KeyError` if *key* is not in the underlying mapping.

iter(proxy)

Return an iterator over the keys of the underlying mapping. This is a shortcut for `iter(proxy.keys())`.

len(proxy)

Return the number of items in the underlying mapping.

copy()

Return a shallow copy of the underlying mapping.

get(*key*[, *default*])

Return the value for *key* if *key* is in the underlying mapping, else *default*. If *default* is not given, it defaults to `None`, so that this method never raises a `KeyError`.

items()

Return a new view of the underlying mapping's items (`(key, value)` pairs).

keys()

Return a new view of the underlying mapping's keys.

values()

Return a new view of the underlying mapping's values.

reversed(proxy)

Return a reverse iterator over the keys of the underlying mapping.

Ajouté dans la version 3.9.

hash(proxy)

Return a hash of the underlying mapping.

Ajouté dans la version 3.12.

class `types.`**`CapsuleType`**

The type of capsule objects.

Ajouté dans la version 3.13.

8.10.3 Additional Utility Classes and Functions

class `types.`**`SimpleNamespace`**

A simple `object` subclass that provides attribute access to its namespace, as well as a meaningful repr.

Unlike *object*, with `SimpleNamespace` you can add and remove attributes.

SimpleNamespace objects may be initialized in the same way as *dict* : either with keyword arguments, with a single positional argument, or with both. When initialized with keyword arguments, those are directly added to the underlying namespace. Alternatively, when initialized with a positional argument, the underlying namespace will be updated with key-value pairs from that argument (either a mapping object or an *iterable* object producing key-value pairs). All such keys must be strings.

The type is roughly equivalent to the following code :

```
class SimpleNamespace:
    def __init__(self, mapping_or_iterable=(), /, **kwargs):
        self.__dict__.update(mapping_or_iterable)
        self.__dict__.update(kwargs)

    def __repr__(self):
        items = (f"{k}={v!r}" for k, v in self.__dict__.items())
        return "{}({})".format(type(self).__name__, ", ".join(items))

    def __eq__(self, other):
        if isinstance(self, SimpleNamespace) and isinstance(other,␣
↪SimpleNamespace):
            return self.__dict__ == other.__dict__
        return NotImplemented
```

`SimpleNamespace` may be useful as a replacement for `class NS: pass`. However, for a structured record type use *namedtuple()* instead.

`SimpleNamespace` objects are supported by *copy.replace()*.

Ajouté dans la version 3.3.

Modifié dans la version 3.9 : Attribute order in the repr changed from alphabetical to insertion (like `dict`).

Modifié dans la version 3.13 : Added support for an optional positional argument.

`types.`**`DynamicClassAttribute`**(*fget=None*, *fset=None*, *fdel=None*, *doc=None*)

Route attribute access on a class to __getattr__.

This is a descriptor, used to define attributes that act differently when accessed through an instance and through a class. Instance access remains normal, but access to an attribute through a class will be routed to the class's __getattr__ method ; this is done by raising AttributeError.

This allows one to have properties active on an instance, and have virtual attributes on the class with the same name (see *enum.Enum* for an example).

Ajouté dans la version 3.4.

8.10.4 Coroutine Utility Functions

`types.`**`coroutine`**(*gen_func*)

This function transforms a *generator* function into a *coroutine function* which returns a generator-based coroutine. The generator-based coroutine is still a *generator iterator*, but is also considered to be a *coroutine* object and is *awaitable*. However, it may not necessarily implement the `__await__()` method.

If *gen_func* is a generator function, it will be modified in-place.

If *gen_func* is not a generator function, it will be wrapped. If it returns an instance of *collections.abc.Generator*, the instance will be wrapped in an *awaitable* proxy object. All other types of objects will be returned as is.

Ajouté dans la version 3.5.

8.11 copy --- Shallow and deep copy operations

Code source : Lib/copy.py

Les instructions d'affectation en Python ne copient pas les objets, elles créent des liens entre la cible et l'objet. Concernant les collections qui sont mutables ou contiennent des éléments mutables, une copie est parfois nécessaire, pour pouvoir modifier une copie sans modifier l'autre. Ce module met à disposition des opérations de copie génériques superficielle et récursive (comme expliqué ci-dessous).

Résumé de l'interface :

`copy.copy(obj)`

Return a shallow copy of *obj*.

`copy.deepcopy(obj[, memo])`

Return a deep copy of *obj*.

`copy.replace(obj, /, **changes)`

Creates a new object of the same type as *obj*, replacing fields with values from *changes*.

Ajouté dans la version 3.13.

exception `copy.Error`

Levée pour les erreurs spécifiques au module.

La différence entre copie superficielle et récursive n'est pertinente que pour les objets composés (objets contenant d'autres objets, comme des listes ou des instances de classe) :

- Une *copie superficielle* construit un nouvel objet composé puis (dans la mesure du possible) insère dans l'objet composé des *références* aux objets trouvés dans l'original.
- Une *copie récursive (ou profonde)* construit un nouvel objet composé puis, récursivement, insère dans l'objet composé des *copies* des objets trouvés dans l'objet original.

On rencontre souvent deux problèmes avec les opérations de copie récursive qui n'existent pas avec les opérations de copie superficielle :

- Les objets récursifs (objets composés qui, directement ou indirectement, contiennent une référence à eux-mêmes) peuvent causer une boucle récursive.
- Comme une copie récursive copie tout, elle peut en copier trop, par exemple des données qui sont destinées à être partagées entre différentes copies.

La fonction `deepcopy()` évite ces problèmes en :

- gardant en mémoire dans un dictionnaire `memo` les objets déjà copiés durant la phase de copie actuelle ; et
- laissant les classes créées par l'utilisateur écraser l'opération de copie ou l'ensemble de composants copiés.

This module does not copy types like module, method, stack trace, stack frame, file, socket, window, or any similar types. It does "copy" functions and classes (shallow and deeply), by returning the original object unchanged ; this is compatible with the way these are treated by the `pickle` module.

Les copies superficielles de dictionnaires peuvent être faites en utilisant `dict.copy()`, et de listes en affectant un `slice` de la liste, par exemple, `copied_list = original_list[:]`.

Les classes peuvent utiliser les mêmes interfaces de contrôle que celles utilisées pour la sérialisation. Voir la description du module `pickle` pour plus d'informations sur ces méthodes. En effet, le module `copy` utilise les fonctions de sérialisation enregistrées à partir du module `copyreg`.

In order for a class to define its own copy implementation, it can define special methods `__copy__()` and `__deepcopy__()`.

`object.__copy__(self)`

Called to implement the shallow copy operation ; no additional arguments are passed.

`object.__deepcopy__(self, memo)`

Called to implement the deep copy operation ; it is passed one argument, the *memo* dictionary. If the `__deepcopy__` implementation needs to make a deep copy of a component, it should call the `deepcopy()` function with the component as first argument and the *memo* dictionary as second argument. The *memo* dictionary should be treated as an opaque object.

Function `copy.replace()` is more limited than `copy()` and `deepcopy()`, and only supports named tuples created by `namedtuple()`, `dataclasses`, and other classes which define method `__replace__()`.

`object.__replace__(self, /, **changes)`

This method should create a new object of the same type, replacing fields with values from *changes*.

Voir aussi

Module `pickle`
Discussion sur les méthodes spéciales utilisées pour gérer la récupération et la restauration de l'état d'un objet.

8.12 `pprint` — L'affichage élégant de données

Code source : Lib/pprint.py

Le module `pprint` permet « d'afficher élégamment » des structures de données Python arbitraires sous une forme qui peut être utilisée ensuite comme une entrée dans l'interpréteur. Si les structures formatées incluent des objets qui ne sont pas des types Python fondamentaux, leurs représentations peuvent ne pas être acceptables en tant que telles par l'interpréteur. Cela peut être le cas si des objets tels que des fichiers, des interfaces de connexion (*sockets* en anglais) ou des classes sont inclus, c'est aussi valable pour beaucoup d'autres types d'objets qui ne peuvent être représentés sous forme littérale en Python.

L'affichage formaté affiche, tant que possible, les objets sur une seule ligne et les sépare sur plusieurs lignes s'ils dépassent la largeur autorisée par l'interpréteur. Cette largeur est ajustable par le paramètre *width* qui est par défaut de 80 caractères.

Les dictionnaires sont classés par clés avant que l'affichage ne soit calculé.

Modifié dans la version 3.9 : Prise en charge de l'affichage élégant de `types.SimpleNamespace`.

Modifié dans la version 3.10 : Prise en charge de l'affichage élégant de `dataclasses.dataclass`.

8.12.1 Fonctions

`pprint.pp(object, stream=None, indent=1, width=80, depth=None, *, compact=False, sort_dicts=False, underscore_numbers=False)`

Affiche la représentation formatée de *object*, suivie d'un retour à la ligne. Cette fonction peut être utilisée dans l'interpréteur interactif au lieu de la fonction `print()` pour inspecter les valeurs. Astuce : vous pouvez réassigner `print = pprint.pp` pour l'utiliser dans une portée.

Paramètres

- **`object`** -- L'objet à afficher.
- **`stream`** (*file-like object* | None) -- Un objet fichier-compatible sur lequel la sortie sera écrite en appelant sa méthode `write()`. Si la valeur est `None` (la valeur par défaut), `sys.stdout` est utilisé.
- **`indent`** (`int`) -- La quantité d'indentation ajoutée pour chaque niveau d'imbrication.
- **`width`** (`int`) -- Le nombre maximal de caractères souhaités par ligne dans la sortie. Si une structure ne peut pas être formatée dans la largeur limite donnée, un effort maximal sera fait.
- **`depth`** (`int | None`) -- Le nombre de niveaux d'imbrication qui peuvent être affichés. Si la structure de données à afficher est trop profonde, le niveau suivant est remplacé par `....` Si la valeur est `None` (la valeur par défaut), il n'y a pas de contrainte sur la profondeur des objets formatés.
- **`compact`** (`bool`) -- Contrôle la façon dont les longues *sequences* sont formatées. Si la valeur est `False` (la valeur par défaut), chaque élément d'une séquence sera formaté sur une ligne séparée, sinon autant d'éléments que possible qui peuvent tenir dans la *width* (la largeur) seront formatés sur chaque ligne de sortie.

— **sort_dicts** (bool) -- Si la valeur est True, les dictionnaires seront formatés avec leurs clés triées, sinon ils seront affichés dans l'ordre d'insertion (la valeur par défaut).
— **underscore_numbers** (bool) -- Si la valeur est True, les entiers seront formatés avec le caractère _ comme séparateur de milliers, sinon les underscores ne sont pas affichés (le comportement par défaut).

```
>>> import pprint
>>> stuff = ['spam', 'eggs', 'lumberjack', 'knights', 'ni']
>>> stuff.insert(0, stuff)
>>> pprint.pp(stuff)
[<Recursion on list with id=...>,
 'spam',
 'eggs',
 'lumberjack',
 'knights',
 'ni']
```

Ajouté dans la version 3.8.

pprint.**pprint**(*object*, *stream=None*, *indent=1*, *width=80*, *depth=None*, ***, *compact=False*, *sort_dicts=True*, *underscore_numbers=False*)

Alias pour *pp()* avec *sort_dicts* initialisé à True par défaut (ce qui trie automatiquement les clés des dictionnaires), vous pourriez préférer utiliser *pp()* à la place où *sort_dicts* est initialisé à False par défaut.

pprint.**pformat**(*object*, *indent=1*, *width=80*, *depth=None*, ***, *compact=False*, *sort_dicts=True*, *underscore_numbers=False*)

Renvoie une représentation formatée de *object* sous forme de chaîne de caractères. *indent*, *width*, *depth*, *compact*, *sort_dicts* et *underscore_numbers* sont passés au constructeur de *PrettyPrinter* comme paramètres de formatage et leurs significations sont telles que décrites dans la documentation ci-dessus.

pprint.**isreadable**(*object*)

Détermine si la représentation formatée de *object* est « lisible », ou s'il peut être utilisé pour recomposer sa valeur en utilisant la fonction *eval()*. Cela renvoie toujours False pour les objets récursifs.

```
>>> pprint.isreadable(stuff)
False
```

pprint.**isrecursive**(*object*)

Détermine si *object* nécessite une représentation récursive. Cette fonction est soumise aux mêmes limitations que celles notées dans *saferepr()* ci-dessous et peut lever une exception *RecursionError* si elle échoue à détecter un objet récursif.

pprint.**saferepr**(*object*)

Renvoie une représentation de *object*, protégée contre la récursivité dans certaines structures de données courantes, à savoir que les instances de *dict*, *list* et *tuple* ou les sous-classes dont la méthode __repr__ n'ont pas été surchargées. Si la représentation de *object* présente une entrée récursive, celle-ci sera représentée telle que <Recursion on typename with id=number>. Par ailleurs, la représentation de l'objet n'est pas formatée.

```
>>> pprint.saferepr(stuff)
"[<Recursion on list with id=...>, 'spam', 'eggs', 'lumberjack', 'knights', 'ni
↪']"
```

8.12.2 Les Objets PrettyPrinter

class pprint.**PrettyPrinter**(*indent=1*, *width=80*, *depth=None*, *stream=None*, ***, *compact=False*, *sort_dicts=True*, *underscore_numbers=False*)

Crée une instance de *PrettyPrinter*.

Les arguments ont la même signification que pour *pp()*. Notez qu'ils sont dans un ordre différent, et que *sort_dicts* est initialisé à `True` par défaut.

```
>>> import pprint
>>> stuff = ['spam', 'eggs', 'lumberjack', 'knights', 'ni']
>>> stuff.insert(0, stuff[:])
>>> pp = pprint.PrettyPrinter(indent=4)
>>> pp.pprint(stuff)
[   ['spam', 'eggs', 'lumberjack', 'knights', 'ni'],
    'spam',
    'eggs',
    'lumberjack',
    'knights',
    'ni']
>>> pp = pprint.PrettyPrinter(width=41, compact=True)
>>> pp.pprint(stuff)
[['spam', 'eggs', 'lumberjack',
  'knights', 'ni'],
 'spam', 'eggs', 'lumberjack', 'knights',
 'ni']
>>> tup = ('spam', ('eggs', ('lumberjack', ('knights', ('ni', ('dead',
... ('parrot', ('fresh fruit',))))))))
>>> pp = pprint.PrettyPrinter(depth=6)
>>> pp.pprint(tup)
('spam', ('eggs', ('lumberjack', ('knights', ('ni', ('dead', (...)))))))
```

Modifié dans la version 3.4 : Ajout du paramètre *compact*.

Modifié dans la version 3.8 : Ajout du paramètre *sort_dicts*.

Modifié dans la version 3.10 : Ajout du paramètre *underscore_numbers*.

Modifié dans la version 3.11 : N'essaie plus d'écrire sur `sys.stdout` lorsqu'il vaut `None`.

Les instances de la classe *PrettyPrinter* ont les méthodes suivantes :

`PrettyPrinter.`**`pformat`**(*object*)

Renvoie la représentation formatée de *object*. Cela prend en compte les options passées au constructeur de la classe *PrettyPrinter*.

`PrettyPrinter.`**`pprint`**(*object*)

Affiche sur le flux configuré la représentation formatée de *object*, suivie d'une fin de ligne.

Les méthodes suivantes fournissent les implémentations pour les fonctions correspondantes de mêmes noms. L'utilisation de ces méthodes sur une instance est légèrement plus efficace, car les nouveaux objets *PrettyPrinter* n'ont pas besoin d'être créés.

`PrettyPrinter.`**`isreadable`**(*object*)

Détermine si la représentation formatée de *object* est « lisible », ou si elle peut être utilisée pour recomposer sa valeur en utilisant la fonction *eval()*. Cela renvoie toujours `False` pour les objets récursifs. Si le paramètre *depth* de la classe *PrettyPrinter* est initialisé et que l'objet est plus « profond » que permis, cela renvoie `False`.

`PrettyPrinter.`**`isrecursive`**(*object*)

Détermine si l'objet nécessite une représentation récursive.

Cette méthode est fournie sous forme de point d'entrée ou méthode (à déclenchement) automatique (*hook* en anglais) pour permettre aux sous-classes de modifier la façon dont les objets sont convertis en chaînes. L'implémentation par défaut est celle de la fonction *saferepr()*.

`PrettyPrinter.`**`format`**(*object*, *context*, *maxlevels*, *level*)

Renvoie trois valeurs : la version formatée de *object* sous forme de chaîne de caractères, une option indiquant si le résultat est « lisible », et une option indiquant si une récursion a été détectée. Le premier argument est

l'objet à représenter. Le deuxième est un dictionnaire qui contient l'`id()` des objets (conteneurs directs ou indirects de *objet* qui affectent sa représentation) qui font partie du contexte de représentation courant tel que les clés ; si un objet doit être représenté, mais l'a déjà été dans ce contexte, le troisième argument renvoie `True`. Des appels récursifs à la méthode `format()` doivent ajouter des entrés additionnelles aux conteneurs de ce dictionnaire. Le troisième argument *maxlevels*, donne la limite maximale de récursivité ; la valeur par défaut est `0`. Cet argument doit être passé non modifié pour des appels non récursifs. Le quatrième argument, *level*, donne le niveau de récursivité courant ; les appels récursifs doivent être passés à une valeur inférieure à celle de l'appel courant.

8.12.3 Exemple

Pour illustrer quelques cas pratiques de l'utilisation de la fonction `pp()` et de ses paramètres, allons chercher des informations sur un projet PyPI :

```
>>> import json
>>> import pprint
>>> from urllib.request import urlopen
>>> with urlopen('https://pypi.org/pypi/sampleproject/1.2.0/json') as resp:
...     project_info = json.load(resp)['info']
```

Dans sa forme basique, la fonction `pp()` affiche l'intégralité de l'objet :

```
>>> pprint.pp(project_info)
{'author': 'The Python Packaging Authority',
 'author_email': 'pypa-dev@googlegroups.com',
 'bugtrack_url': None,
 'classifiers': ['Development Status :: 3 - Alpha',
                 'Intended Audience :: Developers',
                 'License :: OSI Approved :: MIT License',
                 'Programming Language :: Python :: 2',
                 'Programming Language :: Python :: 2.6',
                 'Programming Language :: Python :: 2.7',
                 'Programming Language :: Python :: 3',
                 'Programming Language :: Python :: 3.2',
                 'Programming Language :: Python :: 3.3',
                 'Programming Language :: Python :: 3.4',
                 'Topic :: Software Development :: Build Tools'],
 'description': 'A sample Python project\n'
                '=======================\n'
                '\n'
                'This is the description file for the project.\n'
                '\n'
                'The file should use UTF-8 encoding and be written using '
                'ReStructured Text. It\n'
                'will be used to generate the project webpage on PyPI, and '
                'should be written for\n'
                'that purpose.\n'
                '\n'
                'Typical contents for this file would include an overview of '
                'the project, basic\n'
                'usage examples, etc. Generally, including the project '
                'changelog in here is not\n'
                'a good idea, although a simple "What\'s New" section for the '
                'most recent version\n'
                'may be appropriate.',
 'description_content_type': None,
 'docs_url': None,
 'download_url': 'UNKNOWN',
```

(suite sur la page suivante)

(suite de la page précédente)

```
 'downloads': {'last_day': -1, 'last_month': -1, 'last_week': -1},
 'home_page': 'https://github.com/pypa/sampleproject',
 'keywords': 'sample setuptools development',
 'license': 'MIT',
 'maintainer': None,
 'maintainer_email': None,
 'name': 'sampleproject',
 'package_url': 'https://pypi.org/project/sampleproject/',
 'platform': 'UNKNOWN',
 'project_url': 'https://pypi.org/project/sampleproject/',
 'project_urls': {'Download': 'UNKNOWN',
                  'Homepage': 'https://github.com/pypa/sampleproject'},
 'release_url': 'https://pypi.org/project/sampleproject/1.2.0/',
 'requires_dist': None,
 'requires_python': None,
 'summary': 'A sample Python project',
 'version': '1.2.0'}
```

Le résultat peut être limité à une certaine profondeur en initialisant *depth*. (... est utilisé pour des contenus plus « profonds ») :

```
>>> pprint.pp(project_info, depth=1)
{'author': 'The Python Packaging Authority',
 'author_email': 'pypa-dev@googlegroups.com',
 'bugtrack_url': None,
 'classifiers': [...],
 'description': 'A sample Python project\n'
                '=======================\n'
                '\n'
                'This is the description file for the project.\n'
                '\n'
                'The file should use UTF-8 encoding and be written using '
                'ReStructured Text. It\n'
                'will be used to generate the project webpage on PyPI, and '
                'should be written for\n'
                'that purpose.\n'
                '\n'
                'Typical contents for this file would include an overview of '
                'the project, basic\n'
                'usage examples, etc. Generally, including the project '
                'changelog in here is not\n'
                'a good idea, although a simple "What\'s New" section for the '
                'most recent version\n'
                'may be appropriate.',
 'description_content_type': None,
 'docs_url': None,
 'download_url': 'UNKNOWN',
 'downloads': {...},
 'home_page': 'https://github.com/pypa/sampleproject',
 'keywords': 'sample setuptools development',
 'license': 'MIT',
 'maintainer': None,
 'maintainer_email': None,
 'name': 'sampleproject',
 'package_url': 'https://pypi.org/project/sampleproject/',
 'platform': 'UNKNOWN',
```

(suite sur la page suivante)

(suite de la page précédente)

```
 'project_url': 'https://pypi.org/project/sampleproject/',
 'project_urls': {...},
 'release_url': 'https://pypi.org/project/sampleproject/1.2.0/',
 'requires_dist': None,
 'requires_python': None,
 'summary': 'A sample Python project',
 'version': '1.2.0'}
```

De plus, une valeur maximale de caractères sur une ligne peut être définie en initialisant le paramètre *width*. Si un long objet ne peut être scindé, la valeur donnée à *width* sera outrepassée :

```
>>> pprint.pp(project_info, depth=1, width=60)
{'author': 'The Python Packaging Authority',
 'author_email': 'pypa-dev@googlegroups.com',
 'bugtrack_url': None,
 'classifiers': [...],
 'description': 'A sample Python project\n'
                '========================\n'
                '\n'
                'This is the description file for the '
                'project.\n'
                '\n'
                'The file should use UTF-8 encoding and be '
                'written using ReStructured Text. It\n'
                'will be used to generate the project '
                'webpage on PyPI, and should be written '
                'for\n'
                'that purpose.\n'
                '\n'
                'Typical contents for this file would '
                'include an overview of the project, '
                'basic\n'
                'usage examples, etc. Generally, including '
                'the project changelog in here is not\n'
                'a good idea, although a simple "What\'s '
                'New" section for the most recent version\n'
                'may be appropriate.',
 'description_content_type': None,
 'docs_url': None,
 'download_url': 'UNKNOWN',
 'downloads': {...},
 'home_page': 'https://github.com/pypa/sampleproject',
 'keywords': 'sample setuptools development',
 'license': 'MIT',
 'maintainer': None,
 'maintainer_email': None,
 'name': 'sampleproject',
 'package_url': 'https://pypi.org/project/sampleproject/',
 'platform': 'UNKNOWN',
 'project_url': 'https://pypi.org/project/sampleproject/',
 'project_urls': {...},
 'release_url': 'https://pypi.org/project/sampleproject/1.2.0/',
 'requires_dist': None,
 'requires_python': None,
 'summary': 'A sample Python project',
 'version': '1.2.0'}
```

8.13 reprlib --- Alternate repr() implementation

Code source : Lib/reprlib.py

The reprlib module provides a means for producing object representations with limits on the size of the resulting strings. This is used in the Python debugger and may be useful in other contexts as well.

This module provides a class, an instance, and a function :

class reprlib.**Repr**(*, *maxlevel=6*, *maxtuple=6*, *maxlist=6*, *maxarray=5*, *maxdict=4*, *maxset=6*, *maxfrozenset=6*, *maxdeque=6*, *maxstring=30*, *maxlong=40*, *maxother=30*, *fillvalue='...'*, *indent=None*)

Class which provides formatting services useful in implementing functions similar to the built-in repr() ; size limits for different object types are added to avoid the generation of representations which are excessively long.

The keyword arguments of the constructor can be used as a shortcut to set the attributes of the Repr instance. Which means that the following initialization :

```
aRepr = reprlib.Repr(maxlevel=3)
```

est équivalent à :

```
aRepr = reprlib.Repr()
aRepr.maxlevel = 3
```

See section *Repr Objects* for more information about Repr attributes.

Modifié dans la version 3.12 : Allow attributes to be set via keyword arguments.

reprlib.**aRepr**

This is an instance of Repr which is used to provide the repr() function described below. Changing the attributes of this object will affect the size limits used by repr() and the Python debugger.

reprlib.**repr**(*obj*)

This is the repr() method of aRepr. It returns a string similar to that returned by the built-in function of the same name, but with limits on most sizes.

In addition to size-limiting tools, the module also provides a decorator for detecting recursive calls to __repr__() and substituting a placeholder string instead.

@reprlib.**recursive_repr**(*fillvalue='...'*)

Decorator for __repr__() methods to detect recursive calls within the same thread. If a recursive call is made, the *fillvalue* is returned, otherwise, the usual __repr__() call is made. For example :

```
>>> from reprlib import recursive_repr
>>> class MyList(list):
...     @recursive_repr()
...     def __repr__(self):
...         return '<' + '|'.join(map(repr, self)) + '>'
...
>>> m = MyList('abc')
>>> m.append(m)
>>> m.append('x')
>>> print(m)
<'a'|'b'|'c'|...|'x'>
```

Ajouté dans la version 3.2.

8.13.1 Repr Objects

Repr instances provide several attributes which can be used to provide size limits for the representations of different object types, and methods which format specific object types.

Repr.**fillvalue**

This string is displayed for recursive references. It defaults to `....`

Ajouté dans la version 3.11.

Repr.**maxlevel**

Depth limit on the creation of recursive representations. The default is `6`.

Repr.**maxdict**

Repr.**maxlist**

Repr.**maxtuple**

Repr.**maxset**

Repr.**maxfrozenset**

Repr.**maxdeque**

Repr.**maxarray**

Limits on the number of entries represented for the named object type. The default is `4` for *maxdict*, `5` for *maxarray*, and `6` for the others.

Repr.**maxlong**

Maximum number of characters in the representation for an integer. Digits are dropped from the middle. The default is `40`.

Repr.**maxstring**

Limit on the number of characters in the representation of the string. Note that the "normal" representation of the string is used as the character source : if escape sequences are needed in the representation, these may be mangled when the representation is shortened. The default is `30`.

Repr.**maxother**

This limit is used to control the size of object types for which no specific formatting method is available on the *Repr* object. It is applied in a similar manner as *maxstring*. The default is `20`.

Repr.**indent**

If this attribute is set to `None` (the default), the output is formatted with no line breaks or indentation, like the standard *repr()*. For example :

```
>>> example = [
...     1, 'spam', {'a': 2, 'b': 'spam eggs', 'c': {3: 4.5, 6: []}}, 'ham']
>>> import reprlib
>>> aRepr = reprlib.Repr()
>>> print(aRepr.repr(example))
[1, 'spam', {'a': 2, 'b': 'spam eggs', 'c': {3: 4.5, 6: []}}, 'ham']
```

If *indent* is set to a string, each recursion level is placed on its own line, indented by that string :

```
>>> aRepr.indent = '-->'
>>> print(aRepr.repr(example))
[
-->1,
-->'spam',
-->{
-->-->'a': 2,
-->-->'b': 'spam eggs',
-->-->'c': {
-->-->-->3: 4.5,
```

(suite sur la page suivante)

(suite de la page précédente)

```
-->-->-->6: [],
-->-->},
-->},
-->'ham',
]
```

Setting `indent` to a positive integer value behaves as if it was set to a string with that number of spaces :

```
>>> aRepr.indent = 4
>>> print(aRepr.repr(example))
[
    1,
    'spam',
    {
        'a': 2,
        'b': 'spam eggs',
        'c': {
            3: 4.5,
            6: [],
        },
    },
    'ham',
]
```

Ajouté dans la version 3.12.

`Repr.`**`repr`**(*obj*)

The equivalent to the built-in `repr()` that uses the formatting imposed by the instance.

`Repr.`**`repr1`**(*obj*, *level*)

Recursive implementation used by `repr()`. This uses the type of *obj* to determine which formatting method to call, passing it *obj* and *level*. The type-specific methods should call `repr1()` to perform recursive formatting, with `level - 1` for the value of *level* in the recursive call.

`Repr.`**`repr_TYPE`**(*obj*, *level*)

Formatting methods for specific types are implemented as methods with a name based on the type name. In the method name, **TYPE** is replaced by `'_'.join(type(obj).__name__.split())`. Dispatch to these methods is handled by `repr1()`. Type-specific methods which need to recursively format a value should call `self.repr1(subobj, level - 1)`.

8.13.2 Subclassing Repr Objects

The use of dynamic dispatching by `Repr.repr1()` allows subclasses of `Repr` to add support for additional built-in object types or to modify the handling of types already supported. This example shows how special support for file objects could be added :

```
import reprlib
import sys

class MyRepr(reprlib.Repr):

    def repr_TextIOWrapper(self, obj, level):
        if obj.name in {'<stdin>', '<stdout>', '<stderr>'}:
            return obj.name
        return repr(obj)

aRepr = MyRepr()
print(aRepr.repr(sys.stdin))          # prints '<stdin>'
```

```
<stdin>
```

8.14 enum --- Support for enumerations

Ajouté dans la version 3.4.

Code source : Lib/enum.py

> **Important**
>
> Cette page contient les informations de référence de l'API. Pour des informations sur le didacticiel et une discussion sur des sujets plus avancés, reportez-vous à
> — Tutoriel de base
> — Tutoriel avancé
> — Recettes pour les énumérations

Une énumération :
— est un ensemble de noms symboliques (appelés membres) liés à des valeurs uniques,
— can be iterated over to return its canonical (i.e. non-alias) members in definition order
— utilise la syntaxe *d'appel* pour renvoyer les valeurs de ses membres,
— utilise la syntaxe *d'indiçage* pour renvoyer les noms de ses membres.

Les énumérations sont créées soit en utilisant la syntaxe `class`, soit en utilisant la syntaxe d'appel de fonction :

```
>>> from enum import Enum

>>> # class syntax
>>> class Color(Enum):
...     RED = 1
...     GREEN = 2
...     BLUE = 3

>>> # functional syntax
>>> Color = Enum('Color', [('RED', 1), ('GREEN', 2), ('BLUE', 3)])
```

Même si on peut utiliser la syntaxe `class` pour créer des énumérations, les *Enums* ne sont pas des vraies classes Python. Lisez En quoi les énumérations sont-elles différentes ? pour plus de détails.

> **Note**
>
> Nomenclature
> — The class `Color` is an *enumeration* (or *enum*)
> — The attributes `Color.RED`, `Color.GREEN`, etc., are *enumeration members* (or *members*) and are functionally constants.
> — The enum members have *names* and *values* (the name of `Color.RED` is `RED`, the value of `Color.BLUE` is `3`, etc.)

8.14.1 Contenu du module

EnumType

Le `type` d'*Enum* et de ses sous-classes.

Enum

Classe mère pour créer une énumération de constantes.

IntEnum

Classe mère pour créer une énumération de constantes qui sont également des sous-classes de `int` (*notes*).

StrEnum

Classe mère pour créer une énumération de constantes qui sont également des sous-classes de `str` (*notes*).

Flag

Classe mère pour créer une énumération de constantes pouvant être combinées avec des opérateurs de comparaison bit-à-bit, sans perdre leur qualité de `Flag`.

IntFlag

Classe mère pour créer une énumération de constantes pouvant être combinées avec des opérateurs de comparaison bit-à-bit, sans perdre leur qualité de `IntFlag`. Les membres de `IntFlag` sont aussi des sous-classes de `int` (*notes*).

ReprEnum

Used by `IntEnum`, `StrEnum`, and `IntFlag` to keep the `str()` of the mixed-in type.

EnumCheck

Énumération avec les valeurs `CONTINUOUS`, `NAMED_FLAGS` et `UNIQUE`, à utiliser avec `verify()` pour s'assurer que diverses contraintes sont satisfaites par une énumération donnée.

FlagBoundary

Énumération avec les valeurs `STRICT`, `CONFORM`, `EJECT` et `KEEP` qui permet un contrôle plus précis sur la façon dont les valeurs invalides sont traitées dans une énumération.

EnumDict

A subclass of `dict` for use when subclassing `EnumType`.

auto

Les instances sont remplacées par une valeur appropriée pour les membres de l'énumération. `StrEnum` utilise par défaut la version minuscule du nom du membre, tandis que les autres énumérations utilisent 1 par défaut et puis augmentent.

property()

Allows `Enum` members to have attributes without conflicting with member names. The `value` and `name` attributes are implemented this way.

unique()

Décorateur de classe qui garantit qu'une valeur ne puisse être associée qu'à un seul nom.

verify()

Décorateur de classe qui vérifie des contraintes personnalisées pour une énumération.

member()

Fait de `obj` un membre. Peut être utilisé comme décorateur.

nonmember()

Fait que `obj` n'est pas un membre. Peut être utilisé comme décorateur.

global_enum()

Modify the `str()` and `repr()` of an enum to show its members as belonging to the module instead of its class, and export the enum members to the global namespace.

show_flag_values()

Return a list of all power-of-two integers contained in a flag.

Ajouté dans la version 3.6 : `Flag`, `IntFlag`, `auto`

Ajouté dans la version 3.11 : `StrEnum`, `EnumCheck`, `ReprEnum`, `FlagBoundary`, `property`, `member`, `nonmember`, `global_enum`, `show_flag_values`

Ajouté dans la version 3.13 : `EnumDict`

8.14.2 Types de données

class enum.**EnumType**

EnumType est la *métaclasse* pour les énumérations. Il est possible de sous-classer *EnumType* — voir Sous-classer EnumType pour plus de détails.

`EnumType` is responsible for setting the correct `__repr__()`, `__str__()`, `__format__()`, and `__reduce__()` methods on the final *enum*, as well as creating the enum members, properly handling duplicates, providing iteration over the enum class, etc.

__call__(*cls*, *value*, *names=None*, **, module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Cette méthode peut être appelée de deux manières différentes :

— pour chercher un membre existant :

cls
Classe de l'énumération concernée.

value
Valeur à chercher.

— to use the `cls` enum to create a new enum (only if the existing enum does not have any members) :

cls
Classe de l'énumération concernée.

value
Nom de la nouvelle énumération à créer.

names
Couples nom-valeur des membres de la nouvelle énumération.

module
Nom du module dans lequel la classe *Enum* se trouve.

qualname
Position dans le module où la classe *Enum* se trouve.

type
Type à mélanger pour la nouvelle énumération.

start
The first integer value for the Enum (used by `auto`).

boundary
How to handle out-of-range values from bit operations (`Flag` only).

__contains__(*cls*, *member*)

Renvoie `True` si le membre appartient à `cls` :

```
>>> some_var = Color.RED
>>> some_var in Color
True
>>> Color.RED.value in Color
True
```

Modifié dans la version 3.12 : Before Python 3.12, a `TypeError` is raised if a non-Enum-member is used in a containment check.

__dir__(*cls*)

Renvoie `['__class__', '__doc__', '__members__', '__module__']` et les noms des membres de *cls* :

```
>>> dir(Color)
['BLUE', 'GREEN', 'RED', '__class__', '__contains__', '__doc__', '__
↪getitem__', '__init_subclass__', '__iter__', '__len__', '__members__', '_
↪_module__', '__name__', '__qualname__']
```

__getitem__(*cls*, *name*)

Returns the Enum member in *cls* matching *name*, or raises a `KeyError` :

```
>>> Color['BLUE']
<Color.BLUE: 3>
```

__iter__(*cls*)

Renvoie chaque membre de *cls* dans l'ordre de définition :

```
>>> list(Color)
[<Color.RED: 1>, <Color.GREEN: 2>, <Color.BLUE: 3>]
```

__len__(*cls*)

Renvoie le nombre de membres de *cls* :

```
>>> len(Color)
3
```

__members__

Returns a mapping of every enum name to its member, including aliases

__reversed__(*cls*)

Renvoie chaque membre de *cls* dans l'ordre inverse de définition :

```
>>> list(reversed(Color))
[<Color.BLUE: 3>, <Color.GREEN: 2>, <Color.RED: 1>]
```

_add_alias_()

Adds a new name as an alias to an existing member. Raises a `NameError` if the name is already assigned to a different member.

_add_value_alias_()

Adds a new value as an alias to an existing member. Raises a `ValueError` if the value is already linked with a different member.

Ajouté dans la version 3.11 : Before 3.11 `EnumType` was called `EnumMeta`, which is still available as an alias.

class enum.**Enum**

Enum est la classe mère de toutes les énumérations.

name

Le nom utilisé pour définir le membre de l'`Enum` :

```
>>> Color.BLUE.name
'BLUE'
```

value

La valeur attribuée au membre de l'`Enum` :

```
>>> Color.RED.value
1
```

Value of the member, can be set in `__new__()`.

> **Note**
>
> Valeurs des membres d'une *Enum*
>
> Member values can be anything : `int`, `str`, etc. If the exact value is unimportant you may use `auto` instances and an appropriate value will be chosen for you. See `auto` for the details.

> While mutable/unhashable values, such as `dict`, `list` or a mutable `dataclass`, can be used, they will have a quadratic performance impact during creation relative to the total number of mutable/unhashable values in the enum.

name

Name of the member.

value

Value of the member, can be set in `__new__()`.

order

No longer used, kept for backward compatibility. (class attribute, removed during class creation).

ignore

`_ignore_` n'est utilisé que lors de la création et est supprimé de l'énumération une fois la création terminée.

`_ignore_` est une liste de noms qui ne deviendront pas membres et qui seront également supprimés de l'énumération terminée. Voir Intervalle de temps pour un exemple.

__dir__(*self*)

Renvoie `['__class__', '__doc__', '__module__', 'name', 'value']` et toutes les méthodes publiques définies pour *self.__class__* :

```
>>> from datetime import date
>>> class Weekday(Enum):
...     MONDAY = 1
...     TUESDAY = 2
...     WEDNESDAY = 3
...     THURSDAY = 4
...     FRIDAY = 5
...     SATURDAY = 6
...     SUNDAY = 7
...     @classmethod
...     def today(cls):
...         print('today is %s' % cls(date.today().isoweekday()).name)
...
>>> dir(Weekday.SATURDAY)
['__class__', '__doc__', '__eq__', '__hash__', '__module__', 'name', 'today
↪', 'value']
```

_generate_next_value_(*name*, *start*, *count*, *last_values*)

name
Nom du membre en cours de définition (par ex. `RED`).

start
Valeur de départ pour l'énumération ; 1 par défaut.

count
Nombre de membres actuellement définis, le membre actuel n'étant pas inclus.

last_values
Liste des valeurs précédentes.

Méthode statique utilisée pour déterminer la prochaine valeur à renvoyer par `auto` :

```
>>> from enum import auto
>>> class PowersOfThree(Enum):
...     @staticmethod
...     def _generate_next_value_(name, start, count, last_values):
...         return 3 ** (count + 1)
```

(suite sur la page suivante)

(suite de la page précédente)

```
...     FIRST = auto()
...     SECOND = auto()
...
>>> PowersOfThree.SECOND.value
9
```

__init__(*self*, **args*, ***kwds*)

By default, does nothing. If multiple values are given in the member assignment, those values become separate arguments to `__init__`; e.g.

```
>>> from enum import Enum
>>> class Weekday(Enum):
...     MONDAY = 1, 'Mon'
```

`Weekday.__init__()` would be called as `Weekday.__init__(self, 1, 'Mon')`

__init_subclass__(*cls*, ***kwds*)

Méthode de classe utilisée pour personnaliser davantage les sous-classes à venir. Ne fait rien par défaut.

missing(*cls*, *value*)

Méthode de classe pour chercher des valeurs non trouvées dans *cls*. Ne fait rien par défaut mais peut être surchargée pour implémenter un comportement personnalisé de recherche :

```
>>> from enum import StrEnum
>>> class Build(StrEnum):
...     DEBUG = auto()
...     OPTIMIZED = auto()
...     @classmethod
...     def _missing_(cls, value):
...         value = value.lower()
...         for member in cls:
...             if member.value == value:
...                 return member
...         return None
...
>>> Build.DEBUG.value
'debug'
>>> Build('deBUG')
<Build.DEBUG: 'debug'>
```

__new__(*cls*, **args*, ***kwds*)

By default, doesn't exist. If specified, either in the enum class definition or in a mixin class (such as `int`), all values given in the member assignment will be passed; e.g.

```
>>> from enum import Enum
>>> class MyIntEnum(int, Enum):
...     TWENTYSIX = '1a', 16
```

results in the call `int('1a', 16)` and a value of `26` for the member.

> **Note**
>
> When writing a custom `__new__`, do not use `super().__new__` -- call the appropriate `__new__` instead.

__repr__(*self*)

Renvoie la chaîne utilisée pour les appels à *repr()*. Par défaut, renvoie le nom de l'énumération, le nom

du membre et sa valeur, mais peut être surchargée :

```
>>> class OtherStyle(Enum):
...     ALTERNATE = auto()
...     OTHER = auto()
...     SOMETHING_ELSE = auto()
...     def __repr__(self):
...         cls_name = self.__class__.__name__
...         return f'{cls_name}.{self.name}'
...
>>> OtherStyle.ALTERNATE, str(OtherStyle.ALTERNATE), f"{OtherStyle.
↪ALTERNATE}"
(OtherStyle.ALTERNATE, 'OtherStyle.ALTERNATE', 'OtherStyle.ALTERNATE')
```

__str__(*self*)

Renvoie la chaîne utilisée pour les appels à *str()*. Par défaut, renvoie le nom *Enum* et le nom du membre, mais peut être remplacé :

```
>>> class OtherStyle(Enum):
...     ALTERNATE = auto()
...     OTHER = auto()
...     SOMETHING_ELSE = auto()
...     def __str__(self):
...         return f'{self.name}'
...
>>> OtherStyle.ALTERNATE, str(OtherStyle.ALTERNATE), f"{OtherStyle.
↪ALTERNATE}"
(<OtherStyle.ALTERNATE: 1>, 'ALTERNATE', 'ALTERNATE')
```

__format__(*self*)

Returns the string used for *format()* and *f-string* calls. By default, returns `__str__()` return value, but can be overridden :

```
>>> class OtherStyle(Enum):
...     ALTERNATE = auto()
...     OTHER = auto()
...     SOMETHING_ELSE = auto()
...     def __format__(self, spec):
...         return f'{self.name}'
...
>>> OtherStyle.ALTERNATE, str(OtherStyle.ALTERNATE), f"{OtherStyle.
↪ALTERNATE}"
(<OtherStyle.ALTERNATE: 1>, 'OtherStyle.ALTERNATE', 'ALTERNATE')
```

Note

L'utilisation de `auto` avec `Enum` donne des nombres entiers de valeur croissante, en commençant par `1`.

Modifié dans la version 3.12 : Added enum-dataclass-support

class enum.**IntEnum**

IntEnum is the same as `Enum`, but its members are also integers and can be used anywhere that an integer can be used. If any integer operation is performed with an *IntEnum* member, the resulting value loses its enumeration status.

```
>>> from enum import IntEnum
>>> class Number(IntEnum):
```

(suite sur la page suivante)

(suite de la page précédente)

```
...     ONE = 1
...     TWO = 2
...     THREE = 3
...
>>> Number.THREE
<Number.THREE: 3>
>>> Number.ONE + Number.TWO
3
>>> Number.THREE + 5
8
>>> Number.THREE == 3
True
```

Note

L'utilisation de `auto` avec `IntEnum` donne des entiers de valeur croissante, en commençant par `1`.

Modifié dans la version 3.11 : `__str__()` is now `int.__str__()` to better support the *replacement of existing constants* use-case. `__format__()` was already `int.__format__()` for that same reason.

class `enum.`**`StrEnum`**

`StrEnum` is the same as `Enum`, but its members are also strings and can be used in most of the same places that a string can be used. The result of any string operation performed on or with a *StrEnum* member is not part of the enumeration.

Note

There are places in the stdlib that check for an exact `str` instead of a `str` subclass (i.e. `type(unknown) == str` instead of `isinstance(unknown, str)`), and in those locations you will need to use `str(StrEnum.member)`.

Note

L'utilisation de `auto` avec `StrEnum` donne le nom du membre en minuscule comme valeur.

Note

`__str__()` is `str.__str__()` to better support the *replacement of existing constants* use-case. `__format__()` is likewise `str.__format__()` for that same reason.

Ajouté dans la version 3.11.

class `enum.`**`Flag`**

`Flag` is the same as `Enum`, but its members support the bitwise operators `&` (*AND*), `|` (*OR*), `^` (*XOR*), and `~` (*INVERT*) ; the results of those operations are (aliases of) members of the enumeration.

`__contains__`(*self*, *value*)

Renvoie *True* si la valeur est dans *self* :

```
>>> from enum import Flag, auto
>>> class Color(Flag):
...     RED = auto()
```

(suite sur la page suivante)

(suite de la page précédente)

```
...     GREEN = auto()
...     BLUE = auto()
...
>>> purple = Color.RED | Color.BLUE
>>> white = Color.RED | Color.GREEN | Color.BLUE
>>> Color.GREEN in purple
False
>>> Color.GREEN in white
True
>>> purple in white
True
>>> white in purple
False
```

__iter__(self):

Returns all contained non-alias members :

```
>>> list(Color.RED)
[<Color.RED: 1>]
>>> list(purple)
[<Color.RED: 1>, <Color.BLUE: 4>]
```

Ajouté dans la version 3.11.

__len__(self):

Renvoie le nombre de membres de *Flag* :

```
>>> len(Color.GREEN)
1
>>> len(white)
3
```

Ajouté dans la version 3.11.

__bool__(self):

Renvoie *True* s'il y a un membre dans les bits de *Flag*, *False* sinon :

```
>>> bool(Color.GREEN)
True
>>> bool(white)
True
>>> black = Color(0)
>>> bool(black)
False
```

__or__(*self*, *other*)

Renvoie le *OU* logique entre le membre et *other* :

```
>>> Color.RED | Color.GREEN
<Color.RED|GREEN: 3>
```

__and__(*self*, *other*)

Renvoie le *ET* logique entre le membre et *other* :

```
>>> purple & white
<Color.RED|BLUE: 5>
>>> purple & Color.GREEN
<Color: 0>
```

__xor__(*self*, *other*)

Renvoie le *OU Exclusif* logique entre le membre et *other* :

```
>>> purple ^ white
<Color.GREEN: 2>
>>> purple ^ Color.GREEN
<Color.RED|GREEN|BLUE: 7>
```

__invert__(self):

Returns all the flags in *type(self)* that are not in *self* :

```
>>> ~white
<Color: 0>
>>> ~purple
<Color.GREEN: 2>
>>> ~Color.RED
<Color.GREEN|BLUE: 6>
```

_numeric_repr_()

Fonction utilisée pour formater toutes les valeurs numériques sans nom restantes. La valeur par défaut est la représentation de la valeur ; les choix courants sont `hex()` et `oct()`.

> **Note**
>
> L'utilisation de `auto` avec `Flag` donne des entiers qui sont des puissances de deux, en commençant par `1`.

Modifié dans la version 3.11 : The *repr()* of zero-valued flags has changed. It is now :

```
>>> Color(0)
<Color: 0>
```

class enum.**IntFlag**

`IntFlag` is the same as `Flag`, but its members are also integers and can be used anywhere that an integer can be used.

```
>>> from enum import IntFlag, auto
>>> class Color(IntFlag):
...     RED = auto()
...     GREEN = auto()
...     BLUE = auto()
...
>>> Color.RED & 2
<Color: 0>
>>> Color.RED | 2
<Color.RED|GREEN: 3>
```

Si une opération sur un entier est effectuée avec un membre d'un *IntFlag*, le résultat n'est pas un *IntFlag* :

```
>>> Color.RED + 2
3
```

If a `Flag` operation is performed with an *IntFlag* member and :
— le résultat est un *IntFlag* valide : un *IntFlag* est renvoyé ;
— the result is not a valid *IntFlag* : the result depends on the `FlagBoundary` setting
The `repr()` of unnamed zero-valued flags has changed. It is now :

```
>>> Color(0)
<Color: 0>
```

Note

L'utilisation de *auto* avec *IntFlag* donne des entiers qui sont des puissances de deux, en commençant par 1.

Modifié dans la version 3.11 : `__str__()` is now `int.__str__()` to better support the *replacement of existing constants* use-case. `__format__()` was already `int.__format__()` for that same reason.

Inversion of an `IntFlag` now returns a positive value that is the union of all flags not in the given flag, rather than a negative value. This matches the existing *Flag* behavior.

class enum.**ReprEnum**

`ReprEnum` uses the *repr()* of *Enum*, but the *str()* of the mixed-in data type :
— `int.__str__()` for *IntEnum* and *IntFlag*
— `str.__str__()` for *StrEnum*
Inherit from `ReprEnum` to keep the *str()* / *format()* of the mixed-in data type instead of using the *Enum*-default *str()*.

Ajouté dans la version 3.11.

class enum.**EnumCheck**

EnumCheck contient les options utilisées par le décorateur *verify()* pour assurer diverses contraintes ; si une contrainte n'est pas validée, une *ValueError* est levée.

UNIQUE

Assure que chaque valeur n'a qu'un seul nom :

```
>>> from enum import Enum, verify, UNIQUE
>>> @verify(UNIQUE)
... class Color(Enum):
...     RED = 1
...     GREEN = 2
...     BLUE = 3
...     CRIMSON = 1
Traceback (most recent call last):
...
ValueError: aliases found in <enum 'Color'>: CRIMSON -> RED
```

CONTINUOUS

Assure qu'il n'y a pas de valeurs manquantes entre le membre ayant la valeur la plus faible et le membre ayant la valeur la plus élevée :

```
>>> from enum import Enum, verify, CONTINUOUS
>>> @verify(CONTINUOUS)
... class Color(Enum):
...     RED = 1
...     GREEN = 2
...     BLUE = 5
Traceback (most recent call last):
...
ValueError: invalid enum 'Color': missing values 3, 4
```

NAMED_FLAGS

Ensure that any flag groups/masks contain only named flags -- useful when values are specified instead of being generated by *auto()* :

```
>>> from enum import Flag, verify, NAMED_FLAGS
>>> @verify(NAMED_FLAGS)
... class Color(Flag):
...     RED = 1
...     GREEN = 2
...     BLUE = 4
...     WHITE = 15
...     NEON = 31
Traceback (most recent call last):
...
ValueError: invalid Flag 'Color': aliases WHITE and NEON are missing␣
↪combined values of 0x18 [use enum.show_flag_values(value) for details]
```

Note

CONTINUOUS et NAMED_FLAGS sont conçus pour fonctionner avec des membres à valeur entière.

Ajouté dans la version 3.11.

class enum.**FlagBoundary**

`FlagBoundary` controls how out-of-range values are handled in *Flag* and its subclasses.

STRICT

Out-of-range values cause a *ValueError* to be raised. This is the default for *Flag* :

```
>>> from enum import Flag, STRICT, auto
>>> class StrictFlag(Flag, boundary=STRICT):
...     RED = auto()
...     GREEN = auto()
...     BLUE = auto()
...
>>> StrictFlag(2**2 + 2**4)
Traceback (most recent call last):
...
ValueError: <flag 'StrictFlag'> invalid value 20
    given 0b0 10100
  allowed 0b0 00111
```

CONFORM

Out-of-range values have invalid values removed, leaving a valid *Flag* value :

```
>>> from enum import Flag, CONFORM, auto
>>> class ConformFlag(Flag, boundary=CONFORM):
...     RED = auto()
...     GREEN = auto()
...     BLUE = auto()
...
>>> ConformFlag(2**2 + 2**4)
<ConformFlag.BLUE: 4>
```

EJECT

Out-of-range values lose their *Flag* membership and revert to *int*.

```
>>> from enum import Flag, EJECT, auto
>>> class EjectFlag(Flag, boundary=EJECT):
...     RED = auto()
```

(suite sur la page suivante)

(suite de la page précédente)

```
...     GREEN = auto()
...     BLUE = auto()
...
>>> EjectFlag(2**2 + 2**4)
20
```

KEEP

Out-of-range values are kept, and the *Flag* membership is kept. This is the default for *IntFlag* :

```
>>> from enum import Flag, KEEP, auto
>>> class KeepFlag(Flag, boundary=KEEP):
...     RED = auto()
...     GREEN = auto()
...     BLUE = auto()
...
>>> KeepFlag(2**2 + 2**4)
<KeepFlag.BLUE|16: 20>
```

Ajouté dans la version 3.11.

class enum.**EnumDict**

EnumDict is a subclass of *dict* that is used as the namespace for defining enum classes (see prepare). It is exposed to allow subclasses of *EnumType* with advanced behavior like having multiple values per member. It should be called with the name of the enum class being created, otherwise private names and internal classes will not be handled correctly.

Note that only the *MutableMapping* interface (`__setitem__()` and *update()*) is overridden. It may be possible to bypass the checks using other `dict` operations like `|=`.

member_names

A list of member names.

Ajouté dans la version 3.13.

Noms de la forme `__dunder__` disponibles

__members__ is a read-only ordered mapping of `member_name` :`member` items. It is only available on the class.

__new__(), if specified, must create and return the enum members ; it is also a very good idea to set the member's `_value_` appropriately. Once all the members are created it is no longer used.

Noms de la forme `_sunder_` disponibles

- *_add_alias_()* -- adds a new name as an alias to an existing member.
- *_add_value_alias_()* -- adds a new value as an alias to an existing member.
- *_name_* -- name of the member
- *_value_* -- value of the member ; can be set in `__new__`
- *_missing_()* -- a lookup function used when a value is not found ; may be overridden
- *_ignore_* -- a list of names, either as a *list* or a *str*, that will not be transformed into members, and will be removed from the final class
- *_order_* -- no longer used, kept for backward compatibility (class attribute, removed during class creation)
- *_generate_next_value_()* -- used to get an appropriate value for an enum member ; may be overridden

> **Note**
>
> For standard *Enum* classes the next value chosen is the highest value seen incremented by one.
>
> For *Flag* classes the next value chosen will be the next highest power-of-two.

— While _sunder_ names are generally reserved for the further development of the *Enum* class and can not be used, some are explicitly allowed :
  — _repr_* (e.g. _repr_html_), as used in IPython's rich display

Ajouté dans la version 3.6 : _missing_, _order_, _generate_next_value_

Ajouté dans la version 3.7 : _ignore_

Ajouté dans la version 3.13 : _add_alias_, _add_value_alias_, _repr_*

8.14.3 Utilitaires et décorateurs

class enum.**auto**

auto can be used in place of a value. If used, the *Enum* machinery will call an *Enum*'s *_generate_next_value_()* to get an appropriate value. For *Enum* and *IntEnum* that appropriate value will be the last value plus one ; for *Flag* and *IntFlag* it will be the first power-of-two greater than the highest value ; for *StrEnum* it will be the lower-cased version of the member's name. Care must be taken if mixing *auto()* with manually specified values.

auto instances are only resolved when at the top level of an assignment :
— FIRST = auto() will work (auto() is replaced with 1) ;
— SECOND = auto(), -2 will work (auto is replaced with 2, so 2, -2 is used to create the SECOND enum member ;
— THREE = [auto(), -3] will *not* work (<auto instance>, -3 is used to create the THREE enum member)

Modifié dans la version 3.11.1 : In prior versions, auto() had to be the only thing on the assignment line to work properly.

_generate_next_value_ peut être surchargée pour personnaliser les valeurs produites par *auto*.

> **Note**
>
> in 3.13 the default _generate_next_value_ will always return the highest member value incremented by 1, and will fail if any member is an incompatible type.

@enum.**property**

Décorateur similaire au *property* natif, mais spécifique aux énumérations. Il permet aux attributs de membre d'avoir les mêmes noms que les membres eux-mêmes.

> **Note**
>
> *property* et le membre doivent être définis dans des classes distinctes ; par exemple, les attributs *value* et *name* sont définis dans la classe *Enum*, et les sous-classes d'*Enum* peuvent définir des membres avec les noms value et name.

Ajouté dans la version 3.11.

@enum.**unique**

A class decorator specifically for enumerations. It searches an enumeration's *__members__*, gathering any aliases it finds ; if any are found *ValueError* is raised with the details :

```
>>> from enum import Enum, unique
>>> @unique
... class Mistake(Enum):
...     ONE = 1
...     TWO = 2
```

(suite sur la page suivante)

(suite de la page précédente)

```
...     THREE = 3
...     FOUR = 3
...
Traceback (most recent call last):
...
ValueError: duplicate values found in <enum 'Mistake'>: FOUR -> THREE
```

`@enum.verify`

Décorateur de `classe` spécifique aux énumérations. Utilisé pour spécifier quelles contraintes doivent être vérifiées par les membres de l'`EnumCheck` décorée.

Ajouté dans la version 3.11.

`@enum.member`

Décorateur à utiliser dans les énumérations : sa cible devient un membre.

Ajouté dans la version 3.11.

`@enum.nonmember`

Décorateur à utiliser dans les énumérations : sa cible ne devient pas un membre.

Ajouté dans la version 3.11.

`@enum.global_enum`

A decorator to change the `str()` and `repr()` of an enum to show its members as belonging to the module instead of its class. Should only be used when the enum members are exported to the module global namespace (see `re.RegexFlag` for an example).

Ajouté dans la version 3.11.

`enum.show_flag_values`(*value*)

Return a list of all power-of-two integers contained in a flag *value*.

Ajouté dans la version 3.11.

8.14.4 Notes

`IntEnum`, `StrEnum` et `IntFlag`

Ces trois types d'énumération sont conçus pour remplacer directement les valeurs existantes basées sur des entiers et des chaînes ; en tant que tels, ils ont des limitations supplémentaires :

— `__str__` utilise la valeur et pas le nom du membre de l'énumération

— `__format__`, parce qu'elle fait appel à `__str__`, utilise également la valeur du membre de l'énumération au lieu de son nom

Si ces limitations ne vous conviennent pas, vous pouvez créer votre propre classe mère en mélangeant vous-même le type `int` ou `str` :

```
>>> from enum import Enum
>>> class MyIntEnum(int, Enum):
...     pass
```

ou vous pouvez réassigner la `str()` appropriée, etc., dans votre énumération :

```
>>> from enum import Enum, IntEnum
>>> class MyIntEnum(IntEnum):
...     __str__ = Enum.__str__
```

8.15 graphlib --- Functionality to operate with graph-like structures

Code source : Lib/graphlib.py

---

class graphlib.**TopologicalSorter**(*graph=None*)

Provides functionality to topologically sort a graph of *hashable* nodes.

L'ordre topologique est un ordre linéaire des sommets d'un graphe afin que pour chaque arête u → v d'un sommet u à un sommet v, cet ordre va placer le sommet u avant le sommet v. Par exemple, les sommets d'un graphe peuvent représenter une tâche à faire et une arête peut représenter la contrainte comme quoi telle tâche doit être réalisée avant telle autre. Dans cet exemple, un ordre topologique est simplement une séquence valide pour ces tâches. Cet ordre n'est possible que si le graphe n'a pas de circuit, c'est-à-dire si c'est un graphe orienté acyclique.

Si l'argument optionnel *graph* est fourni, cela doit être un dictionnaire représentant un graphe acyclique avec comme clés les nœuds et comme valeurs des itérables sur les prédécesseurs de ces nœuds dans le graphe (les nœuds qui ont des arêtes qui pointent vers la valeur de la clé). Les nœuds s'ajoutent en utilisant la méthode `add()`

De manière générale, les étapes nécessaires pour trier un graphe donné sont les suivantes :

— créer une instance de la classe `TopologicalSorter` avec éventuellement un graphe initial ;
— ajouter d'autres nœuds au graphe ;
— appeler `prepare()` sur le graphe ;
— tant que `is_active()` est à `True`, itérer sur les nœuds renvoyés par `get_ready()` pour les traiter. Appeler `done()` sur chaque nœud une fois le traitement terminé.

Si vous souhaitez simplement trier des nœuds du graphe sans parallélisme, la méthode `TopologicalSorter.static_order()` peut être utilisée directement :

```
>>> graph = {"D": {"B", "C"}, "C": {"A"}, "B": {"A"}}
>>> ts = TopologicalSorter(graph)
>>> tuple(ts.static_order())
('A', 'C', 'B', 'D')
```

La classe est conçue pour prendre facilement en charge le traitement en parallèle des nœuds quand ils deviennent disponibles. Par exemple :

```
topological_sorter = TopologicalSorter()

# Add nodes to 'topological_sorter'...

topological_sorter.prepare()
while topological_sorter.is_active():
    for node in topological_sorter.get_ready():
        # Worker threads or processes take nodes to work on off the
        # 'task_queue' queue.
        task_queue.put(node)

    # When the work for a node is done, workers put the node in
    # 'finalized_tasks_queue' so we can get more nodes to work on.
    # The definition of 'is_active()' guarantees that, at this point, at
    # least one node has been placed on 'task_queue' that hasn't yet
    # been passed to 'done()', so this blocking 'get()' must (eventually)
    # succeed.  After calling 'done()', we loop back to call 'get_ready()'
    # again, so put newly freed nodes on 'task_queue' as soon as
    # logically possible.
    node = finalized_tasks_queue.get()
    topological_sorter.done(node)
```

add(*node*, **predecessors*)

Add a new node and its predecessors to the graph. Both the *node* and all elements in *predecessors* must be *hashable*.

S'il est appelé plusieurs fois avec le même nœud en tant qu'argument, l'ensemble des dépendances sera l'union de toutes les dépendances qui auront été transmises.

Il est possible d'ajouter un nœud sans dépendance (*predecessors* n'est pas fourni) ou de fournir une dépendance deux fois. Si un nœud qui n'a jamais été fourni auparavant est inclus dans *predecessors* il sera automatiquement ajouté au graphe sans prédécesseur lui-même.

Lève une `ValueError` si appelée après `prepare()`.

prepare()

Indique que le graphe est terminé et vérifie les circuits du graphe. Si un circuit est détecté, une `CycleError` est levée mais `get_ready()` peut encore être utilisée pour obtenir autant de nœuds que possible avant que les circuits ne bloquent la progression. Après un appel de cette fonction, le graphe ne peut pas être modifié, et donc aucun nœud ne peut être ajouté avec `add()`.

is_active()

Renvoie `True` si une progression peut être faite et `False` dans le cas contraire. La progression est possible si des circuits ne bloquent pas la résolution ou qu'il reste des nœuds prêts qui n'ont pas encore été renvoyés par `TopologicalSorter.get_ready()` ou que le nombre de nœuds marqués `TopologicalSorter.done()` est inférieur au nombre qui a été renvoyé par `TopologicalSorter.get_ready()`.

The `__bool__()` method of this class defers to this function, so instead of :

```
if ts.is_active():
    ...
```

il est plus simple de faire :

```
if ts:
    ...
```

Lève une `ValueError` si l'appel à `prepare()` n'a pas été fait au préalable.

done(**nodes*)

Marque un ensemble de nœuds renvoyé par `TopologicalSorter.get_ready()` comme traités, permettant aux successeurs de chaque nœud de *nodes* d'être renvoyés lors d'un prochain appel à `get_ready()`.

Lève une `ValueError` si n'importe quel nœud dans *nodes* a déjà été marqué comme traité par un précédent appel à cette méthode ou si un nœud n'a pas été ajouté au graphe en utilisant `TopologicalSorter.add()`, si l'appel est fait sans appel à `prepare()` ou si le nœud n'a pas encore été renvoyé par `get_ready()`.

get_ready()

Renvoie un *n*-uplet avec tous les nœuds prêts. Renvoie d'abord tous les nœuds sans prédécesseurs, et une fois marqués comme traités avec un appel de `TopologicalSorter.done()`, les autres appels renvoient tous les nouveaux nœuds dont tous les prédécesseurs sont traités. Une fois que la progression n'est plus possible, des tuples vides sont renvoyés.

Lève une `ValueError` si l'appel à `prepare()` n'a pas été fait au préalable.

static_order()

Returns an iterator object which will iterate over nodes in a topological order. When using this method, `prepare()` and `done()` should not be called. This method is equivalent to :

```
def static_order(self):
    self.prepare()
```

(suite sur la page suivante)

(suite de la page précédente)

```
    while self.is_active():
        node_group = self.get_ready()
        yield from node_group
        self.done(*node_group)
```

Le tri obtenu peut dépendre de l'ordre dans lequel les éléments ont été ajoutés dans le graphe. Par exemple :

```
>>> ts = TopologicalSorter()
>>> ts.add(3, 2, 1)
>>> ts.add(1, 0)
>>> print([*ts.static_order()])
[2, 0, 1, 3]

>>> ts2 = TopologicalSorter()
>>> ts2.add(1, 0)
>>> ts2.add(3, 2, 1)
>>> print([*ts2.static_order()])
[0, 2, 1, 3]
```

Ceci est dû au fait que "0" et "2" sont au même niveau dans le graphe (ils auraient été renvoyés dans le même appel à `get_ready()`) et l'ordre entre eux est déterminé par l'ordre lors de l'insertion.

Si un circuit est détecté alors une `CycleError` est levée.

Ajouté dans la version 3.9.

8.15.1 Exceptions

Le module `graphlib` définit les classes d'exceptions suivantes :

exception graphlib.**CycleError**

Une classe héritant de `ValueError` levée par `TopologicalSorter.prepare()` si un circuit existe dans le graphe courant. Si plusieurs circuits existent, un seul est inclus dans l'exception.

The detected cycle can be accessed via the second element in the `args` attribute of the exception instance and consists in a list of nodes, such that each node is, in the graph, an immediate predecessor of the next node in the list. In the reported list, the first and the last node will be the same, to make it clear that it is cyclic.

CHAPITRE 9

Modules numériques et mathématiques

Les modules documentés dans ce chapitre fournissent des fonctions et des types autour des nombres et des mathématiques. Le module *numbers* définit une hiérarchie abstraite des types numériques. Les modules *math* et *cmath* contiennent des fonctions mathématiques pour les nombres à virgule flottante ou les nombres complexes. Le module *decimal* permet de représenter des nombres décimaux de manière exacte, et utilise une arithmétique de précision arbitraire.

Les modules suivants sont documentés dans ce chapitre :

9.1 numbers --- Numeric abstract base classes

Code source : Lib/numbers.py

The `numbers` module (**PEP 3141**) defines a hierarchy of numeric *abstract base classes* which progressively define more operations. None of the types defined in this module are intended to be instantiated.

class numbers.**Number**

La base de la hiérarchie numérique. Si vous voulez juste vérifier qu'un argument *x* est un nombre, peu importe le type, utilisez `isinstance(x, Number)`.

9.1.1 La tour numérique

class numbers.**Complex**

Subclasses of this type describe complex numbers and include the operations that work on the built-in *complex* type. These are : conversions to *complex* and *bool*, *real*, *imag*, `+`, `-`, `*`, `/`, `**`, *abs()*, *conjugate()*, `==`, and `!=`. All except `-` and `!=` are abstract.

real

Abstrait. Récupère la partie réelle de ce nombre.

imag

Abstrait. Retrouve la partie imaginaire de ce nombre.

abstractmethod conjugate()

Abstrait. Renvoie le complexe conjugué. Par exemple, `(1+3j).conjugate() == (1-3j)`.

class numbers.**Real**

To *Complex*, Real adds the operations that work on real numbers.

En bref, celles-ci sont : une conversion vers *float*, *math.trunc()*, *round()*, *math.floor()*, *math.ceil()*, *divmod()*, //, %, <, <=, > et >=.

Real fournit également des valeurs par défaut pour *complex()*, *real*, *imag* et *conjugate()*.

class numbers.**Rational**

Subtypes *Real* and adds *numerator* and *denominator* properties. It also provides a default for *float()*.

The *numerator* and *denominator* values should be instances of *Integral* and should be in lowest terms with *denominator* positive.

numerator

Abstrait.

denominator

Abstrait.

class numbers.**Integral**

Subtypes *Rational* and adds a conversion to *int*. Provides defaults for *float()*, *numerator*, and *denominator*. Adds abstract methods for *pow()* with modulus and bit-string operations : <<, >>, &, ^, |, ~.

9.1.2 Notes for type implementers

Implementers should be careful to make equal numbers equal and hash them to the same values. This may be subtle if there are two different extensions of the real numbers. For example, *fractions.Fraction* implements *hash()* as follows :

```
def __hash__(self):
    if self.denominator == 1:
        # Get integers right.
        return hash(self.numerator)
    # Expensive check, but definitely correct.
    if self == float(self):
        return hash(float(self))
    else:
        # Use tuple's hash to avoid a high collision rate on
        # simple fractions.
        return hash((self.numerator, self.denominator))
```

Ajouter plus d'ABC numériques

Il est bien entendu possible de créer davantage d'ABC pour les nombres et cette hiérarchie serait médiocre si elle excluait la possibilité d'en ajouter. Vous pouvez ajouter MyFoo entre *Complex* et *Real* ainsi :

```
class MyFoo(Complex): ...
MyFoo.register(Real)
```

Implémentation des opérations arithmétiques

We want to implement the arithmetic operations so that mixed-mode operations either call an implementation whose author knew about the types of both arguments, or convert both to the nearest built in type and do the operation there. For subtypes of *Integral*, this means that __add__() and __radd__() should be defined as :

```
class MyIntegral(Integral):

    def __add__(self, other):
        if isinstance(other, MyIntegral):
```

(suite sur la page suivante)

(suite de la page précédente)

```
            return do_my_adding_stuff(self, other)
        elif isinstance(other, OtherTypeIKnowAbout):
            return do_my_other_adding_stuff(self, other)
        else:
            return NotImplemented

    def __radd__(self, other):
        if isinstance(other, MyIntegral):
            return do_my_adding_stuff(other, self)
        elif isinstance(other, OtherTypeIKnowAbout):
            return do_my_other_adding_stuff(other, self)
        elif isinstance(other, Integral):
            return int(other) + int(self)
        elif isinstance(other, Real):
            return float(other) + float(self)
        elif isinstance(other, Complex):
            return complex(other) + complex(self)
        else:
            return NotImplemented
```

Il existe 5 cas différents pour une opération de type mixte sur des sous-classes de `Complex`. Nous nous référerons à tout le code ci-dessus qui ne se réfère pas à `MyIntegral` et `OtherTypeIKnowAbout` comme "expression générique". `a` est une instance de `A`, qui est un sous-type de `Complex` (`a : A <: Complex`) et `b : B <: Complex`. Considérons `a + b` :

1. If `A` defines an `__add__()` which accepts `b`, all is well.
2. If `A` falls back to the boilerplate code, and it were to return a value from `__add__()`, we'd miss the possibility that `B` defines a more intelligent `__radd__()`, so the boilerplate should return `NotImplemented` from `__add__()`. (Or `A` may not implement `__add__()` at all.)
3. Then `B`'s `__radd__()` gets a chance. If it accepts `a`, all is well.
4. Si elle fait appel au code générique, il n'y a plus de méthode possible à essayer, c'est donc ici que l'implémentation par défaut intervient.
5. Si `B < : A`, Python essaie `B.__radd__` avant `A.__add__`. C'est valable parce qu'elle est implémentée avec la connaissance de `A`, donc elle peut gérer ces instances avant de déléguer à `Complex`.

If `A <: Complex` and `B <: Real` without sharing any other knowledge, then the appropriate shared operation is the one involving the built in `complex`, and both `__radd__()` s land there, so `a+b == b+a`.

Comme la plupart des opérations sur un type donné seront très similaires, il peut être utile de définir une fonction accessoire qui génère les instances résultantes et inverses d'un opérateur donné. Par exemple, `fractions.Fraction` utilise :

```
def _operator_fallbacks(monomorphic_operator, fallback_operator):
    def forward(a, b):
        if isinstance(b, (int, Fraction)):
            return monomorphic_operator(a, b)
        elif isinstance(b, float):
            return fallback_operator(float(a), b)
        elif isinstance(b, complex):
            return fallback_operator(complex(a), b)
        else:
            return NotImplemented
    forward.__name__ = '__' + fallback_operator.__name__ + '__'
    forward.__doc__ = monomorphic_operator.__doc__

    def reverse(b, a):
        if isinstance(a, Rational):
            # Includes ints.
```

(suite sur la page suivante)

(suite de la page précédente)

```
            return monomorphic_operator(a, b)
        elif isinstance(a, Real):
            return fallback_operator(float(a), float(b))
        elif isinstance(a, Complex):
            return fallback_operator(complex(a), complex(b))
        else:
            return NotImplemented
    reverse.__name__ = '__r' + fallback_operator.__name__ + '__'
    reverse.__doc__ = monomorphic_operator.__doc__

    return forward, reverse

def _add(a, b):
    """a + b"""
    return Fraction(a.numerator * b.denominator +
                    b.numerator * a.denominator,
                    a.denominator * b.denominator)

__add__, __radd__ = _operator_fallbacks(_add, operator.add)

# ...
```

9.2 `math` --- Mathematical functions

Ce module fournit l'accès aux fonctions mathématiques définies par la norme C.

Ces fonctions ne peuvent pas être utilisées avec les nombres complexes ; si vous avez besoin de la prise en charge des nombres complexes, utilisez les fonctions du même nom du module *cmath*. La séparation entre les fonctions qui gèrent les nombres complexes et les autres vient du constat que tous les utilisateurs ne souhaitent pas acquérir le niveau mathématique nécessaire à la compréhension des nombres complexes. Recevoir une exception plutôt qu'un nombre complexe en retour d'une fonction permet au programmeur de déterminer immédiatement comment et pourquoi ce nombre a été généré, avant que celui-ci ne soit passé involontairement en paramètre d'une autre fonction.

Les fonctions suivantes sont fournies dans ce module. Sauf mention contraire explicite, toutes les valeurs de retour sont des flottants.

Number-theoretic functions	
comb(n, k)	Number of ways to choose *k* items from *n* items without repetition and wit
factorial(n)	*n* factorial
*gcd(*integers)*	Greatest common divisor of the integer arguments
isqrt(n)	Integer square root of a nonnegative integer *n*
*lcm(*integers)*	Least common multiple of the integer arguments
perm(n, k)	Number of ways to choose *k* items from *n* items without repetition and wit
Floating point arithmetic	
ceil(x)	Ceiling of *x*, the smallest integer greater than or equal to *x*
fabs(x)	Absolute value of *x*
floor(x)	Floor of *x*, the largest integer less than or equal to *x*
fma(x, y, z)	Fused multiply-add operation : `(x * y) + z`
fmod(x, y)	Remainder of division `x / y`
modf(x)	Fractional and integer parts of *x*
remainder(x, y)	Remainder of *x* with respect to *y*
trunc(x)	Integer part of *x*
Floating point manipulation functions	

suite sur la page

Tableau 1 – suite de la page précédente

`copysign(x, y)`	Magnitude (absolute value) of *x* with the sign of *y*
`frexp(x)`	Mantissa and exponent of *x*
`isclose(a, b, rel_tol, abs_tol)`	Check if the values *a* and *b* are close to each other
`isfinite(x)`	Check if *x* is neither an infinity nor a NaN
`isinf(x)`	Check if *x* is a positive or negative infinity
`isnan(x)`	Check if *x* is a NaN (not a number)
`ldexp(x, i)`	`x * (2**i)`, inverse of function `frexp()`
`nextafter(x, y, steps)`	Floating-point value *steps* steps after *x* towards *y*
`ulp(x)`	Value of the least significant bit of *x*
Power, exponential and logarithmic functions	
`cbrt(x)`	Cube root of *x*
`exp(x)`	*e* raised to the power *x*
`exp2(x)`	*2* raised to the power *x*
`expm1(x)`	*e* raised to the power *x*, minus 1
`log(x, base)`	Logarithm of *x* to the given base (*e* by default)
`log1p(x)`	Natural logarithm of *1+x* (base *e*)
`log2(x)`	Base-2 logarithm of *x*
`log10(x)`	Base-10 logarithm of *x*
`pow(x, y)`	*x* raised to the power *y*
`sqrt(x)`	Square root of *x*
Summation and product functions	
`dist(p, q)`	Euclidean distance between two points *p* and *q* given as an iterable of coor
`fsum(iterable)`	Sum of values in the input *iterable*
`hypot(*coordinates)`	Euclidean norm of an iterable of coordinates
`prod(iterable, start)`	Product of elements in the input *iterable* with a *start* value
`sumprod(p, q)`	Sum of products from two iterables *p* and *q*
Angular conversion	
`degrees(x)`	Convert angle *x* from radians to degrees
`radians(x)`	Convert angle *x* from degrees to radians
Trigonometric functions	
`acos(x)`	Arc cosine of *x*
`asin(x)`	Arc sine of *x*
`atan(x)`	Arc tangent of *x*
`atan2(y, x)`	`atan(y / x)`
`cos(x)`	Cosine of *x*
`sin(x)`	Sine of *x*
`tan(x)`	Tangent of *x*
Hyperbolic functions	
`acosh(x)`	Inverse hyperbolic cosine of *x*
`asinh(x)`	Inverse hyperbolic sine of *x*
`atanh(x)`	Inverse hyperbolic tangent of *x*
`cosh(x)`	Hyperbolic cosine of *x*
`sinh(x)`	Hyperbolic sine of *x*
`tanh(x)`	Hyperbolic tangent of *x*
Special functions	
`erf(x)`	Error function at *x*
`erfc(x)`	Complementary error function at *x*
`gamma(x)`	Gamma function at *x*
`lgamma(x)`	Natural logarithm of the absolute value of the Gamma function at *x*
Constants	
`pi`	$\pi = 3.141592...$
`e`	$e = 2.718281...$
`tau`	$\tau = 2\pi = 6.283185...$
`inf`	Positive infinity
`nan`	"Not a number" (NaN)

9.2.1 Number-theoretic functions

`math.comb(n, k)`

Renvoie le nombre de façons de choisir *k* éléments parmi *n* de manière non-ordonnée et sans répétition.

Vaut `n! / (k! * (n - k)!)` quand `k <= n` et zéro quand `k > n`.

Also called the binomial coefficient because it is equivalent to the coefficient of k-th term in polynomial expansion of `(1 + x)`^{n}.

Lève une `TypeError` si un des paramètres n'est pas un entier. Lève une `ValueError` si un des paramètres est négatif.

Ajouté dans la version 3.8.

`math.factorial(n)`

Return *n* factorial as an integer. Raises `ValueError` if *n* is not integral or is negative.

Modifié dans la version 3.10 : Floats with integral values (like `5.0`) are no longer accepted.

`math.gcd(*integers)`

Return the greatest common divisor of the specified integer arguments. If any of the arguments is nonzero, then the returned value is the largest positive integer that is a divisor of all arguments. If all arguments are zero, then the returned value is `0`. `gcd()` without arguments returns `0`.

Ajouté dans la version 3.5.

Modifié dans la version 3.9 : Added support for an arbitrary number of arguments. Formerly, only two arguments were supported.

`math.isqrt(n)`

Renvoie la racine carrée entière du nombre positif *n*. C'est la partie entière de la valeur exacte de la racine carrée de *n* ou, de manière équivalente, le plus grand entier *a* tel que a*^{2} ≤ *n*.

Pour certaines applications, il est plus pratique d'avoir le plus petit entier *a* tel que $n \leq a^2$ ou, en d'autres termes, la partie entière de la valeur exacte de la racine carrée de *n*. Pour *n* positif, on peut le calculer avec `a = 1 + isqrt(n - 1)`.

Ajouté dans la version 3.8.

`math.lcm(*integers)`

Return the least common multiple of the specified integer arguments. If all arguments are nonzero, then the returned value is the smallest positive integer that is a multiple of all arguments. If any of the arguments is zero, then the returned value is `0`. `lcm()` without arguments returns `1`.

Ajouté dans la version 3.9.

`math.perm(n, k=None)`

Renvoie le nombre de façons de choisir *k* éléments parmi *n* de manière ordonnée sans répétition.

Vaut `n! / (n - k)!` quand `k <= n` et vaut zéro quand `k > n`.

If *k* is not specified or is `None`, then *k* defaults to *n* and the function returns `n!`.

Lève une `TypeError` si un des paramètres n'est pas un entier. Lève une `ValueError` si un des paramètres est négatif.

Ajouté dans la version 3.8.

9.2.2 Floating point arithmetic

`math.ceil(x)`

Return the ceiling of *x*, the smallest integer greater than or equal to *x*. If *x* is not a float, delegates to `x.__ceil__`, which should return an `Integral` value.

`math.fabs(x)`

Renvoie la valeur absolue de *x*.

math.**floor**(*x*)

Return the floor of *x*, the largest integer less than or equal to *x*. If *x* is not a float, delegates to `x.__floor__`, which should return an `Integral` value.

math.**fma**(*x*, *y*, *z*)

Fused multiply-add operation. Return `(x * y) + z`, computed as though with infinite precision and range followed by a single round to the `float` format. This operation often provides better accuracy than the direct expression `(x * y) + z`.

This function follows the specification of the fusedMultiplyAdd operation described in the IEEE 754 standard. The standard leaves one case implementation-defined, namely the result of `fma(0, inf, nan)` and `fma(inf, 0, nan)`. In these cases, `math.fma` returns a NaN, and does not raise any exception.

Ajouté dans la version 3.13.

math.**fmod**(*x*, *y*)

Return the floating-point remainder of `x / y`, as defined by the platform C library function `fmod(x, y)`. Note that the Python expression `x % y` may not return the same result. The intent of the C standard is that `fmod(x, y)` be exactly (mathematically ; to infinite precision) equal to `x - n*y` for some integer *n* such that the result has the same sign as *x* and magnitude less than `abs(y)`. Python's `x % y` returns a result with the sign of *y* instead, and may not be exactly computable for float arguments. For example, `fmod(-1e-100, 1e100)` is `-1e-100`, but the result of Python's `-1e-100 % 1e100` is `1e100-1e-100`, which cannot be represented exactly as a float, and rounds to the surprising `1e100`. For this reason, function `fmod()` is generally preferred when working with floats, while Python's `x % y` is preferred when working with integers.

math.**modf**(*x*)

Renvoie les parties entière et fractionnelle de *x*. Les deux résultats ont le signe de *x* et sont flottants.

Note that `modf()` has a different call/return pattern than its C equivalents : it takes a single argument and return a pair of values, rather than returning its second return value through an 'output parameter' (there is no such thing in Python).

math.**remainder**(*x*, *y*)

Renvoie le reste selon la norme IEEE 754 de *x* par rapport à *y*. Pour *x* fini et *y* fini non nul, il s'agit de la différence `x - n*y`, où `n` est l'entier le plus proche de la valeur exacte du quotient `x / y`. Si `x / y` est exactement à mi-chemin de deux entiers consécutifs, le plus proche entier *pair* est utilisé pour `n`. Ainsi, le reste `r = remainder(x, y)` vérifie toujours `abs(r) <= 0.5 * abs(y)`.

Les cas spéciaux suivent la norme IEEE 754 : en particulier, `remainder(x, math.inf)` vaut *x* pour tout *x* fini, et `remainder(x, 0)` et `remainder(math.inf, x)` lèvent `ValueError` pour tout *x non-NaN*. Si le résultat de l'opération *remainder* est zéro, alors ce zéro aura le même signe que *x*.

On platforms using IEEE 754 binary floating point, the result of this operation is always exactly representable : no rounding error is introduced.

Ajouté dans la version 3.7.

math.**trunc**(*x*)

Return *x* with the fractional part removed, leaving the integer part. This rounds toward 0 : `trunc()` is equivalent to `floor()` for positive *x*, and equivalent to `ceil()` for negative *x*. If *x* is not a float, delegates to `x.__trunc__`, which should return an `Integral` value.

Pour les fonctions `ceil()`, `floor()`, et `modf()`, notez que *tous* les nombres flottants de magnitude suffisamment grande sont des entiers exacts. Les flottants de Python n'ont généralement pas plus de 53 *bits* de précision (tels que le type C `double` de la plate-forme), en quel cas tout flottant *x* tel que `abs(x) >= 2**52` n'a aucun *bit* fractionnel.

9.2.3 Floating point manipulation functions

math.**copysign**(*x*, *y*)

Renvoie un flottant contenant la magnitude (valeur absolue) de *x* mais avec le signe de *y*. Sur les plates-formes prenant en charge les zéros signés, `copysign(1.0, -0.0)` renvoie `-1.0`.

`math.frexp(x)`

Renvoie la mantisse et l'exposant de *x* dans un couple `(m, e)`. *m* est un flottant et *e* est un entier tels que `x == m * 2**e` exactement. Si *x* vaut zéro, renvoie `(0, 0)`, sinon `0.5 <= abs(m) < 1`. Ceci est utilisé pour « extraire » la représentation interne d'un flottant de manière portable.

Note that `frexp()` has a different call/return pattern than its C equivalents : it takes a single argument and return a pair of values, rather than returning its second return value through an 'output parameter' (there is no such thing in Python).

`math.isclose(a, b, *, rel_tol=1e-09, abs_tol=0.0)`

Renvoie `True` si les valeurs *a* et *b* sont proches l'une de l'autre, et `False` sinon.

Whether or not two values are considered close is determined according to given absolute and relative tolerances. If no errors occur, the result will be : `abs(a-b) <= max(rel_tol * max(abs(a), abs(b)), abs_tol)`.

rel_tol is the relative tolerance -- it is the maximum allowed difference between *a* and *b*, relative to the larger absolute value of *a* or *b*. For example, to set a tolerance of 5%, pass `rel_tol=0.05`. The default tolerance is `1e-09`, which assures that the two values are the same within about 9 decimal digits. *rel_tol* must be nonnegative and less than `1.0`.

abs_tol is the absolute tolerance ; it defaults to `0.0` and it must be nonnegative. When comparing `x` to `0.0`, `isclose(x, 0)` is computed as `abs(x) <= rel_tol * abs(x)`, which is `False` for any `x` and rel_tol less than `1.0`. So add an appropriate positive abs_tol argument to the call.

Les valeurs spécifiques suivantes : `NaN`, `inf`, et `-inf` définies dans la norme IEEE 754 seront manipulées selon les règles du standard IEEE. En particulier, `NaN` n'est considéré proche d'aucune autre valeur, `NaN` inclus. `inf` et `-inf` ne sont considérées proches que d'elles-mêmes.

Ajouté dans la version 3.5.

Voir aussi

PEP 485 — Une fonction pour tester des quasi-égalités

`math.isfinite(x)`

Renvoie `True` si *n* n'est ni infini, ni NaN, et `False` sinon. (Notez que `0.0` *est* considéré comme fini.)

Ajouté dans la version 3.2.

`math.isinf(x)`

Renvoie `True` si *x* vaut l'infini positif ou négatif, et `False` sinon.

`math.isnan(x)`

Renvoie `True` si *x* est NaN (*Not a Number*, ou *Pas un Nombre* en français), et `False` sinon.

`math.ldexp(x, i)`

Renvoie `x * (2**i)`. C'est essentiellement l'inverse de la fonction `frexp()`.

`math.nextafter(x, y, steps=1)`

Return the floating-point value *steps* steps after *x* towards *y*.

If *x* is equal to *y*, return *y*, unless *steps* is zero.

Exemples :

- `math.nextafter(x, math.inf)` augmente de valeur : vers l'infini positif.
- `math.nextafter(x, -math.inf)` baisse de valeur : vers l'infini négatif.
- `math.nextafter(x, 0.0)` augmente ou baisse de valeur, vers zéro.
- `math.nextafter(x, math.copysign(math.inf, x))` augmente ou baisse de valeur, en s'éloignant de zéro.

Voir aussi : `math.ulp()`.

Ajouté dans la version 3.9.

Modifié dans la version 3.12 : Added the *steps* argument.

`math.ulp(x)`

Return the value of the least significant bit of the float *x* :

— If *x* is a NaN (not a number), return *x*.
— If *x* is negative, return `ulp(-x)`.
— If *x* is a positive infinity, return *x*.
— If *x* is equal to zero, return the smallest positive *denormalized* representable float (smaller than the minimum positive *normalized* float, `sys.float_info.min`).
— If *x* is equal to the largest positive representable float, return the value of the least significant bit of *x*, such that the first float smaller than *x* is `x - ulp(x)`.
— Otherwise (*x* is a positive finite number), return the value of the least significant bit of *x*, such that the first float bigger than *x* is `x + ulp(x)`.

ULP stands for "Unit in the Last Place".

See also `math.nextafter()` and `sys.float_info.epsilon`.

Ajouté dans la version 3.9.

9.2.4 Power, exponential and logarithmic functions

`math.cbrt(x)`

Return the cube root of *x*.

Ajouté dans la version 3.11.

`math.exp(x)`

Renvoie *e* à la puissance *x*, où *e* = 2.718281… est la base des logarithmes naturels. Cela est en général plus précis que `math.e ** x` ou `pow(math.e, x)`.

`math.exp2(x)`

Return *2* raised to the power *x*.

Ajouté dans la version 3.11.

`math.expm1(x)`

Return *e* raised to the power *x*, minus 1. Here *e* is the base of natural logarithms. For small floats *x*, the subtraction in `exp(x) - 1` can result in a significant loss of precision ; the `expm1()` function provides a way to compute this quantity to full precision :

```
>>> from math import exp, expm1
>>> exp(1e-5) - 1  # gives result accurate to 11 places
1.0000050000069649e-05
>>> expm1(1e-5)    # result accurate to full precision
1.0000050000166668e-05
```

Ajouté dans la version 3.2.

`math.log(x[, base])`

Avec un argument, renvoie le logarithme naturel de *x* (en base *e*).

Avec deux arguments, renvoie le logarithme de *x* en la *base* donnée, calculé par `log(x)/log(base)`.

`math.log1p(x)`

Renvoie le logarithme naturel de *1+x* (en base *e*). Le résultat est calculé par un moyen qui reste exact pour *x* proche de zéro.

`math.log2(x)`

Renvoie le logarithme en base 2 de *x*. C'est en général plus précis que `log(x, 2)`.

Ajouté dans la version 3.3.

> **Voir aussi**
>
> `int.bit_length()` renvoie le nombre de bits nécessaires pour représenter un entier en binaire, en excluant le signe et les zéros de début.

`math.log10(x)`

Renvoie le logarithme de *x* en base 10. C'est habituellement plus exact que `log(x, 10)`.

`math.pow(x, y)`

Return *x* raised to the power *y*. Exceptional cases follow the IEEE 754 standard as far as possible. In particular, `pow(1.0, x)` and `pow(x, 0.0)` always return `1.0`, even when *x* is a zero or a NaN. If both *x* and *y* are finite, *x* is negative, and *y* is not an integer then `pow(x, y)` is undefined, and raises `ValueError`.

À l'inverse de l'opérateur interne `**`, la fonction `math.pow()` convertit ses deux arguments en `float`. Utilisez `**` ou la primitive `pow()` pour calculer des puissances exactes d'entiers.

Modifié dans la version 3.11 : The special cases `pow(0.0, -inf)` and `pow(-0.0, -inf)` were changed to return `inf` instead of raising `ValueError`, for consistency with IEEE 754.

`math.sqrt(x)`

Renvoie la racine carrée de *x*.

9.2.5 Summation and product functions

`math.dist(p, q)`

Renvoie la distance Euclienne entre deux points *p* et *q*, passés comme des séquences (ou des itérables) de coordonnées. Les deux points doivent avoir la même dimension.

À peu près équivalent à :

```
sqrt(sum((px - qx) ** 2.0 for px, qx in zip(p, q)))
```

Ajouté dans la version 3.8.

`math.fsum(iterable)`

Return an accurate floating-point sum of values in the iterable. Avoids loss of precision by tracking multiple intermediate partial sums.

La précision de cet algorithme dépend des garanties arithmétiques de IEEE-754 et des cas standards où le mode d'arrondi est *half-even*. Sur certaines versions non Windows, la bibliothèque C sous-jacente utilise une addition par précision étendue et peut occasionnellement effectuer un double-arrondi sur une somme intermédiaire causant la prise d'une mauvaise valeur du bit de poids faible.

For further discussion and two alternative approaches, see the ASPN cookbook recipes for accurate floating-point summation.

`math.hypot(*coordinates)`

Renvoie la norme Euclidienne, `sqrt(sum(x**2 for x in coordinates))`. C'est la norme du vecteur entre l'origine et le point donné par les coordonnées.

Pour un point bi-dimensionnel `(x, y)`, c'est équivalent à calculer la valeur de l'hypoténuse d'un triangle rectangle en utilisant le théorème de Pythagore, `sqrt(x*x + y*y)`.

Modifié dans la version 3.8 : Ajout de la gestion des points à n-dimensions. Auparavant seuls les points bi-dimensionnels étaient gérés.

Modifié dans la version 3.10 : Improved the algorithm's accuracy so that the maximum error is under 1 ulp (unit in the last place). More typically, the result is almost always correctly rounded to within 1/2 ulp.

`math.prod(iterable, *, start=1)`

Calcule le produit de tous les éléments passés dans l'entrée *iterable*. La valeur de *départ* par défaut du produit vaut `1`.

Quand l'itérable est vide, renvoie la valeur de départ. Cette fonction ne doit être utilisée qu'avec des valeurs numériques et peut rejeter les types non-numériques.

Ajouté dans la version 3.8.

`math.sumprod(p, q)`

Return the sum of products of values from two iterables *p* and *q*.

Raises `ValueError` if the inputs do not have the same length.

À peu près équivalent à :

```
sum(map(operator.mul, p, q, strict=True))
```

For float and mixed int/float inputs, the intermediate products and sums are computed with extended precision.

Ajouté dans la version 3.12.

9.2.6 Conversion angulaire

`math.degrees(x)`

Convertit l'angle *x* de radians en degrés.

`math.radians(x)`

Convertit l'ange *x* de degrés en radians.

9.2.7 Fonctions trigonométriques

`math.acos(x)`

Return the arc cosine of *x*, in radians. The result is between `0` and `pi`.

`math.asin(x)`

Return the arc sine of *x*, in radians. The result is between `-pi/2` and `pi/2`.

`math.atan(x)`

Return the arc tangent of *x*, in radians. The result is between `-pi/2` and `pi/2`.

`math.atan2(y, x)`

Renvoie `atan(y / x)`, en radians. Le résultat est entre `-pi` et `pi`. Le vecteur du plan allant de l'origine vers le point `(x, y)` forme cet angle avec l'axe X positif. L'intérêt de `atan2()` est que le signe des deux entrées est connu. Donc elle peut calculer le bon quadrant pour l'angle. par exemple `atan(1)` et `atan2(1, 1)` donnent tous deux `pi/4`, mais `atan2(-1, -1)` donne `-3*pi/4`.

`math.cos(x)`

Renvoie le cosinus de *x* radians.

`math.sin(x)`

Renvoie le sinus de *x* radians.

`math.tan(x)`

Renvoie la tangente de *x* radians.

9.2.8 Fonctions hyperboliques

Hyperbolic functions are analogs of trigonometric functions that are based on hyperbolas instead of circles.

`math.acosh(x)`

Renvoie l'arc cosinus hyperbolique de *x*.

`math.asinh(x)`

Renvoie l'arc sinus hyperbolique de *x*.

math.**atanh**(*x*)

Renvoie l'arc tangente hyperbolique de *x*.

math.**cosh**(*x*)

Renvoie le cosinus hyperbolique de *x*.

math.**sinh**(*x*)

Renvoie le sinus hyperbolique de *x*.

math.**tanh**(*x*)

Renvoie la tangente hyperbolique de *x*.

9.2.9 Fonctions spéciales

math.**erf**(*x*)

Renvoie la fonction d'erreur en *x*.

The *erf()* function can be used to compute traditional statistical functions such as the cumulative standard normal distribution :

```
def phi(x):
    'Cumulative distribution function for the standard normal distribution'
    return (1.0 + erf(x / sqrt(2.0))) / 2.0
```

Ajouté dans la version 3.2.

math.**erfc**(*x*)

Renvoie la fonction d'erreur complémentaire en *x*. La fonction d'erreur complémentaire est définie par `1.0 - erf(x)`. Elle est utilisée pour les grandes valeurs de *x*, où la soustraction en partant de 1,0 entraînerait une perte de précision.

Ajouté dans la version 3.2.

math.**gamma**(*x*)

Renvoie la fonction Gamma en *x*.

Ajouté dans la version 3.2.

math.**lgamma**(*x*)

Renvoie le logarithme naturel de la valeur absolue de la fonction gamma en *x*.

Ajouté dans la version 3.2.

9.2.10 Constantes

math.**pi**

La constante mathématique $\pi = 3.141592\ldots$, à la précision disponible.

math.**e**

La constante mathématique $e = 2.718281\ldots$, à la précision disponible.

math.**tau**

La constante mathématique $\tau = 6.283185\ldots$, à la précision disponible. Tau est une constante du cercle égale à 2 *π*, le rapport de la circonférence d'un cercle à son rayon. Pour en apprendre plus sur Tau, regardez la vidéo de Vi Hart, Pi is (still) Wrong, et profitez-en pour célébrer le Jour de Tau en bavardant comme deux pies !

Ajouté dans la version 3.6.

math.**inf**

Un flottant positif infini. (Pour un infini négatif, utilisez `-math.inf`.) Équivalent au résultat de `float('inf')`.

Ajouté dans la version 3.5.

`math.nan`

A floating-point "not a number" (NaN) value. Equivalent to the output of `float('nan')`. Due to the requirements of the IEEE-754 standard, `math.nan` and `float('nan')` are not considered to equal to any other numeric value, including themselves. To check whether a number is a NaN, use the *`isnan()`* function to test for NaNs instead of `is` or `==`. Example :

```
>>> import math
>>> math.nan == math.nan
False
>>> float('nan') == float('nan')
False
>>> math.isnan(math.nan)
True
>>> math.isnan(float('nan'))
True
```

Ajouté dans la version 3.5.

Modifié dans la version 3.11 : It is now always available.

Le module *`math`* consiste majoritairement en un conteneur pour les fonctions mathématiques de la bibliothèque C de la plate-forme. Le comportement dans les cas spéciaux suit l'annexe 'F' du standard C99 quand c'est approprié. L'implémentation actuelle lève une *`ValueError`* pour les opérations invalides telles que `sqrt(-1.0)` ou `log(0.0)` (où le standard C99 recommande de signaler que l'opération est invalide ou qu'il y a division par zéro), et une *`OverflowError`* pour les résultats qui débordent (par exemple `exp(1000.0)`). *NaN* ne sera renvoyé pour aucune des fonctions ci-dessus, sauf si au moins un des arguments de la fonction vaut *NaN*. Dans ce cas, la plupart des fonctions renvoient *NaN*, mais (à nouveau, selon l'annexe 'F' du standard C99) il y a quelques exceptions à cette règle, par exemple `pow(float('nan'), 0.0)` ou `hypot(float('nan'), float('inf'))`.

Notez que Python ne fait aucun effort pour distinguer les NaNs signalétiques des NaNs silencieux, et le comportement de signalement des NaNs reste non-spécifié. Le comportement standard est de traiter tous les NaNs comme s'ils étaient silencieux.

Voir aussi

Module *`cmath`*
Version complexe de beaucoup de ces fonctions.

9.3 `cmath` --- Mathematical functions for complex numbers

This module provides access to mathematical functions for complex numbers. The functions in this module accept integers, floating-point numbers or complex numbers as arguments. They will also accept any Python object that has either a `__complex__()` or a `__float__()` method : these methods are used to convert the object to a complex or floating-point number, respectively, and the function is then applied to the result of the conversion.

Note

For functions involving branch cuts, we have the problem of deciding how to define those functions on the cut itself. Following Kahan's "Branch cuts for complex elementary functions" paper, as well as Annex G of C99 and later C standards, we use the sign of zero to distinguish one side of the branch cut from the other : for a branch cut along (a portion of) the real axis we look at the sign of the imaginary part, while for a branch cut along the imaginary axis we look at the sign of the real part.

For example, the *`cmath.sqrt()`* function has a branch cut along the negative real axis. An argument of `-2-0j` is treated as though it lies *below* the branch cut, and so gives a result on the negative imaginary axis :

```
>>> cmath.sqrt(-2-0j)
-1.4142135623730951j
```

But an argument of `-2+0j` is treated as though it lies above the branch cut :

```
>>> cmath.sqrt(-2+0j)
1.4142135623730951j
```

9.3.1 Conversion vers et à partir de coordonnées polaires

A Python complex number `z` is stored internally using *rectangular* or *Cartesian* coordinates. It is completely determined by its *real part* `z.real` and its *imaginary part* `z.imag`.

Les *coordonnées polaires* donnent une manière alternative de représenter un nombre complexe. En coordonnées polaires, un nombre complexe *z* est défini par son module *r* et par son argument (*angle de phase*) *phi*. Le module *r* est la distance entre *z* et l'origine, alors que l'argument *phi* est l'angle (dans le sens inverse des aiguilles d'une montre, ou sens trigonométrique), mesuré en radians, à partir de l'axe X positif, et vers le segment de droite joignant *z* à l'origine.

Les fonctions suivantes peuvent être utilisées pour convertir à partir des coordonnées rectangulaires natives vers les coordonnées polaires, et vice-versa.

`cmath.`**`phase`**(*x*)

Return the phase of *x* (also known as the *argument* of *x*), as a float. `phase(x)` is equivalent to `math.atan2(x.imag, x.real)`. The result lies in the range [-π, π], and the branch cut for this operation lies along the negative real axis. The sign of the result is the same as the sign of `x.imag`, even when `x.imag` is zero :

```
>>> phase(-1+0j)
3.141592653589793
>>> phase(-1-0j)
-3.141592653589793
```

Note

Le module (valeur absolue) d'un nombre complexe *x* peut être calculé en utilisant la primitive `abs()`. Il n'y a pas de fonction spéciale du module `cmath` pour cette opération.

`cmath.`**`polar`**(*x*)

Renvoie la représentation de *x* en coordonnées polaires. Renvoie une paire `(r, phi)` où *r* est le module de *x* et phi est l'argument de *x*. `polar(x)` est équivalent à `(abs(x), phase(x))`.

`cmath.`**`rect`**(*r*, *phi*)

Return the complex number *x* with polar coordinates *r* and *phi*. Equivalent to `complex(r * math.cos(phi), r * math.sin(phi))`.

9.3.2 Fonctions logarithme et exponentielle

`cmath.`**`exp`**(*x*)

Renvoie *e* élevé à la puissance *x*, où *e* est la base des logarithmes naturels.

`cmath.`**`log`**(*x*[, *base*])

Returns the logarithm of *x* to the given *base*. If the *base* is not specified, returns the natural logarithm of *x*. There is one branch cut, from 0 along the negative real axis to -∞.

`cmath.`**`log10`**(*x*)

Renvoie le logarithme en base 10 de *x*. Elle a la même coupure que `log()`.

`cmath.`**`sqrt`**(*x*)

Renvoie la racine carrée de *x*. Elle a la même coupure que `log()`.

9.3.3 Fonctions trigonométriques

`cmath.acos(x)`

Return the arc cosine of *x*. There are two branch cuts : One extends right from 1 along the real axis to ∞. The other extends left from -1 along the real axis to -∞.

`cmath.asin(x)`

Renvoie l'arc sinus de *x*. Elle a les mêmes coupures que `acos()`.

`cmath.atan(x)`

Return the arc tangent of *x*. There are two branch cuts : One extends from `1j` along the imaginary axis to `∞j`. The other extends from `-1j` along the imaginary axis to `-∞j`.

`cmath.cos(x)`

Renvoie le cosinus de *x*.

`cmath.sin(x)`

Renvoie le sinus de *x*.

`cmath.tan(x)`

Renvoie la tangente de *x*.

9.3.4 Fonctions hyperboliques

`cmath.acosh(x)`

Return the inverse hyperbolic cosine of *x*. There is one branch cut, extending left from 1 along the real axis to -∞.

`cmath.asinh(x)`

Return the inverse hyperbolic sine of *x*. There are two branch cuts : One extends from `1j` along the imaginary axis to `∞j`. The other extends from `-1j` along the imaginary axis to `-∞j`.

`cmath.atanh(x)`

Return the inverse hyperbolic tangent of *x*. There are two branch cuts : One extends from `1` along the real axis to ∞. The other extends from `-1` along the real axis to `-∞`.

`cmath.cosh(x)`

Renvoie le cosinus hyperbolique de *x*.

`cmath.sinh(x)`

Renvoie le sinus hyperbolique de *x*.

`cmath.tanh(x)`

Renvoie la tangente hyperbolique de *x*.

9.3.5 Fonctions de classifications

`cmath.isfinite(x)`

Renvoie `True` si la partie réelle *et* la partie imaginaire de *x* sont finies, et `False` sinon.

Ajouté dans la version 3.2.

`cmath.isinf(x)`

Renvoie `True` si soit la partie réelle *ou* la partie imaginaire de *x* est infinie, et `False` sinon.

`cmath.isnan(x)`

Renvoie `True` si soit la partie réelle *ou* la partie imaginaire de *x* est NaN, et `False` sinon.

`cmath.isclose(a, b, *, rel_tol=1e-09, abs_tol=0.0)`

Renvoie `True` si les valeurs *a* et *b* sont proches l'une de l'autre, et `False` sinon.

Whether or not two values are considered close is determined according to given absolute and relative tolerances. If no errors occur, the result will be : `abs(a-b) <= max(rel_tol * max(abs(a), abs(b)), abs_tol)`.

rel_tol is the relative tolerance -- it is the maximum allowed difference between *a* and *b*, relative to the larger absolute value of *a* or *b*. For example, to set a tolerance of 5%, pass `rel_tol=0.05`. The default tolerance is `1e-09`, which assures that the two values are the same within about 9 decimal digits. *rel_tol* must be nonnegative and less than `1.0`.

abs_tol is the absolute tolerance ; it defaults to `0.0` and it must be nonnegative. When comparing `x` to `0.0`, `isclose(x, 0)` is computed as `abs(x) <= rel_tol * abs(x)`, which is `False` for any `x` and rel_tol less than `1.0`. So add an appropriate positive abs_tol argument to the call.

Les valeurs spécifiques suivantes : `NaN`, `inf`, et `-inf` définies dans la norme IEEE 754 seront manipulées selon les règles du standard IEEE. En particulier, `NaN` n'est considéré proche d'aucune autre valeur, `NaN` inclus. `inf` et `-inf` ne sont considérés proches que d'eux-mêmes.

Ajouté dans la version 3.5.

Voir aussi

PEP 485 -- Une fonction pour tester des égalités approximées

9.3.6 Constantes

`cmath.pi`

La constante mathématique π, en tant que flottant.

`cmath.e`

La constante mathématique *e*, en tant que flottant.

`cmath.tau`

La constante mathématique τ, sous forme de flottant.

Ajouté dans la version 3.6.

`cmath.inf`

Nombre à virgule flottante positif infini. Équivaut à `float('inf')`.

Ajouté dans la version 3.6.

`cmath.infj`

Nombre complexe dont la partie réelle vaut zéro et la partie imaginaire un infini positif. Équivalent à `complex(0.0, float('inf'))`.

Ajouté dans la version 3.6.

`cmath.nan`

Un nombre à virgule flottante *NaN* (*Not a number*). Équivalent à `float('nan')`.

Ajouté dans la version 3.6.

`cmath.nanj`

Nombre complexe dont la partie réelle vaut zéro et la partie imaginaire vaut un *NaN*. Équivalent à `complex(0.0, float('nan'))`.

Ajouté dans la version 3.6.

Notez que la sélection de fonctions est similaire, mais pas identique, à celles du module `math`. La raison d'avoir deux modules est que certains utilisateurs ne sont pas intéressés par les nombres complexes, et peut-être ne savent même

pas ce qu'ils sont. Ils préféreraient alors que `math.sqrt(-1)` lève une exception au lieu de renvoyer un nombre complexe. Également, notez que les fonctions définies dans *cmath* renvoient toujours un nombre complexe, même si le résultat peut être exprimé à l'aide d'un nombre réel (en quel cas la partie imaginaire du complexe vaut zéro).

Une note sur les *coupures* : ce sont des courbes sur lesquelles la fonction n'est pas continue. Ce sont des caractéristiques nécessaires de beaucoup de fonctions complexes. Il est supposé que si vous avez besoin d'utiliser des fonctions complexes, vous comprendrez ce que sont les coupures. Consultez n'importe quel livre (pas trop élémentaire) sur les variables complexes pour plus d'informations. Pour des informations sur les choix des coupures à des fins numériques, voici une bonne référence :

Voir aussi

Kahan, W : Branch cuts for complex elementary functions ; or, Much ado about nothing's sign bit. In Iserles, A., and Powell, M. (eds.), The state of the art in numerical analysis. Clarendon Press (1987) pp165--211.

9.4 `decimal` --- Decimal fixed-point and floating-point arithmetic

Code source : Lib/decimal.py

The *decimal* module provides support for fast correctly rounded decimal floating-point arithmetic. It offers several advantages over the *float* datatype :

- Le module `decimal` « est basé sur un modèle en virgule flottante conçu pour les humains, qui suit ce principe directeur : l'ordinateur doit fournir un modèle de calcul qui fonctionne de la même manière que le calcul qu'on apprend à l'école » – extrait (traduit) de la spécification de l'arithmétique décimale.
- Les nombres décimaux peuvent être représentés exactement en base décimale flottante. En revanche, des nombres tels que `1.1` ou `1.2` n'ont pas de représentation exacte en base binaire flottante. L'utilisateur final ne s'attend typiquement pas à obtenir `3.3000000000000003` lorsqu'il saisit `1.1 + 2.2`, ce qui se passe en arithmétique binaire à virgule flottante.
- Ces inexactitudes ont des conséquences en arithmétique. En base décimale à virgule flottante, `0.1 + 0.1 + 0.1 - 0.3` est exactement égal à zéro. En virgule flottante binaire, l'ordinateur l'évalue à `5.5511151231257827e-017`. Bien que très proche de zéro, cette différence induit des erreurs lors des tests d'égalité, erreurs qui peuvent s'accumuler. Pour ces raisons `decimal` est le module utilisé pour des applications comptables ayant des contraintes strictes de fiabilité.
- Le module `decimal` incorpore la notion de chiffres significatifs, de façon à ce que `1.30 + 1.20` égale `2.50`. Le dernier zéro est conservé pour respecter le nombre de chiffres significatifs. C'est l'affichage préféré pour représenter des sommes d'argent. Pour la multiplication, l'approche « scolaire » utilise tous les chiffres présents dans les facteurs. Par exemple, `1.3 * 1.2` donne `1.56` tandis que `1.30 * 1.20` donne `1.5600`.
- Contrairement à l'arithmétique en virgule flottante binaire, le module `decimal` possède un paramètre de précision ajustable (par défaut à 28 chiffres significatifs) qui peut être aussi élevée que nécessaire pour un problème donné :

```
>>> from decimal import *
>>> getcontext().prec = 6
>>> Decimal(1) / Decimal(7)
Decimal('0.142857')
>>> getcontext().prec = 28
>>> Decimal(1) / Decimal(7)
Decimal('0.1428571428571428571428571429')
```

- L'arithmétique binaire et décimale en virgule flottante sont implémentées selon des standards publiés. Alors que le type `float` n'expose qu'une faible portion de ses capacités, le module `decimal` expose tous les composants nécessaires du standard. Lorsque nécessaire, le développeur a un contrôle total de la gestion des signaux et de l'arrondi. Cela inclut la possibilité de forcer une arithmétique exacte en utilisant des exceptions pour bloquer toute opération inexacte.
- Le module `decimal` a été conçu pour gérer « sans préjugé, à la fois une arithmétique décimale non-arrondie (aussi appelée arithmétique en virgule fixe) et à la fois une arithmétique en virgule flottante » (extrait traduit de la spécification de l'arithmétique décimale).

Le module est conçu autour de trois concepts : le nombre décimal, le contexte arithmétique et les signaux.

Un `Decimal` est immuable. Il a un signe, un coefficient et un exposant. Pour préserver le nombre de chiffres significatifs, les zéros en fin de chaîne ne sont pas tronqués. Les décimaux incluent aussi des valeurs spéciales telles que `Infinity`, `-Infinity` et `NaN`. Le standard fait également la différence entre `-0` et `+0`.

Le contexte de l'arithmétique est un environnement qui permet de configurer une précision, une règle pour l'arrondi, des limites sur l'exposant, des options indiquant le résultat des opérations et si les signaux (remontés lors d'opérations illégales) sont traités comme des exceptions Python. Les options d'arrondi incluent *ROUND_CEILING*, *ROUND_DOWN*, *ROUND_FLOOR*, *ROUND_HALF_DOWN*, *ROUND_HALF_EVEN*, *ROUND_HALF_UP*, *ROUND_UP* et *ROUND_05UP*.

Les signaux sont des groupes de conditions exceptionnelles qui surviennent durant le calcul. Selon les besoins de l'application, les signaux peuvent être ignorés, considérés comme de l'information, ou bien traités comme des exceptions. Les signaux dans le module `decimal` sont : *Clamped*, *InvalidOperation*, *DivisionByZero*, *Inexact*, *Rounded*, *Subnormal*, *Overflow*, *Underflow* et *FloatOperation*.

Chaque signal est configurable indépendamment, à travers un drapeau (ou option) et un activateur de déroutement. Quand une opération illégale survient, le drapeau du signal est mis à `1` puis, si l'activateur est configuré, une exception est levée. La mise à `1` du drapeau est persistante, l'utilisateur doit donc remettre les drapeaux à zéro avant de commencer un calcul qu'il souhaite surveiller.

Voir aussi

— Spécification d'IBM sur l'arithmétique décimale : The General Decimal Arithmetic Specification (article en anglais).

9.4.1 Introduction pratique

Commençons par importer le module, regarder le contexte actuel avec *getcontext()* et, si nécessaire, configurer la précision, l'arrondi et la gestion des signaux :

```
>>> from decimal import *
>>> getcontext()
Context(prec=28, rounding=ROUND_HALF_EVEN, Emin=-999999, Emax=999999,
        capitals=1, clamp=0, flags=[], traps=[Overflow, DivisionByZero,
        InvalidOperation])

>>> getcontext().prec = 7       # Set a new precision
```

Les instances de `Decimal` peuvent être construites avec des entiers, des chaînes de caractères, des `floats` ou des *n*-uplets. La construction depuis un entier ou un `float` effectue la conversion exacte de cet entier ou de ce `float`. Les nombres décimaux incluent des valeurs spéciales telles que `NaN` qui signifie en anglais « *Not a number* », en français « pas un nombre », des `Infinity` positifs ou négatifs et `-0` :

```
>>> getcontext().prec = 28
>>> Decimal(10)
Decimal('10')
>>> Decimal('3.14')
Decimal('3.14')
>>> Decimal(3.14)
Decimal('3.14000000000000124344978758017532527446746826171875')
>>> Decimal((0, (3, 1, 4), -2))
Decimal('3.14')
>>> Decimal(str(2.0 ** 0.5))
Decimal('1.4142135623730951')
>>> Decimal(2) ** Decimal('0.5')
Decimal('1.414213562373095048801688724')
>>> Decimal('NaN')
Decimal('NaN')
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> Decimal('-Infinity')
Decimal('-Infinity')
```

Si le signal *FloatOperation* est activé pour déroutement, un mélange accidentel d'objets `Decimal` et de `float` dans les constructeurs ou des opérations de comparaison lève une exception :

```
>>> c = getcontext()
>>> c.traps[FloatOperation] = True
>>> Decimal(3.14)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
decimal.FloatOperation: [<class 'decimal.FloatOperation'>]
>>> Decimal('3.5') < 3.7
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
decimal.FloatOperation: [<class 'decimal.FloatOperation'>]
>>> Decimal('3.5') == 3.5
True
```

Ajouté dans la version 3.3.

Le nombre de chiffres significatifs d'un nouvel objet `Decimal` est déterminé entièrement par le nombre de chiffres saisis. La précision et les règles d'arrondis n'interviennent que lors d'opérations arithmétiques.

```
>>> getcontext().prec = 6
>>> Decimal('3.0')
Decimal('3.0')
>>> Decimal('3.1415926535')
Decimal('3.1415926535')
>>> Decimal('3.1415926535') + Decimal('2.7182818285')
Decimal('5.85987')
>>> getcontext().rounding = ROUND_UP
>>> Decimal('3.1415926535') + Decimal('2.7182818285')
Decimal('5.85988')
```

Si les limites internes de la version en C sont dépassées, la construction d'un objet décimal lève l'exception *InvalidOperation* :

```
>>> Decimal("1e999999999999999999")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
decimal.InvalidOperation: [<class 'decimal.InvalidOperation'>]
```

Modifié dans la version 3.3.

Decimals interact well with much of the rest of Python. Here is a small decimal floating-point flying circus :

```
>>> data = list(map(Decimal, '1.34 1.87 3.45 2.35 1.00 0.03 9.25'.split()))
>>> max(data)
Decimal('9.25')
>>> min(data)
Decimal('0.03')
>>> sorted(data)
[Decimal('0.03'), Decimal('1.00'), Decimal('1.34'), Decimal('1.87'),
 Decimal('2.35'), Decimal('3.45'), Decimal('9.25')]
>>> sum(data)
Decimal('19.29')
>>> a,b,c = data[:3]
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> str(a)
'1.34'
>>> float(a)
1.34
>>> round(a, 1)
Decimal('1.3')
>>> int(a)
1
>>> a * 5
Decimal('6.70')
>>> a * b
Decimal('2.5058')
>>> c % a
Decimal('0.77')
```

Et certaines fonctions mathématiques sont également disponibles sur des instances de `Decimal` :

```
>>> getcontext().prec = 28
>>> Decimal(2).sqrt()
Decimal('1.414213562373095048801688724')
>>> Decimal(1).exp()
Decimal('2.718281828459045235360287471')
>>> Decimal('10').ln()
Decimal('2.302585092994045684017991455')
>>> Decimal('10').log10()
Decimal('1')
```

La méthode *quantize()* arrondit un nombre à un exposant déterminé. Cette méthode est utile pour des applications monétaires qui arrondissent souvent un résultat à un nombre déterminé de chiffres après la virgule :

```
>>> Decimal('7.325').quantize(Decimal('.01'), rounding=ROUND_DOWN)
Decimal('7.32')
>>> Decimal('7.325').quantize(Decimal('1.'), rounding=ROUND_UP)
Decimal('8')
```

Comme montré plus haut, la fonction *getcontext()* accède au contexte actuel et permet de modifier les paramètres. Cette approche répond aux besoins de la plupart des applications.

Pour un travail plus avancé, il peut être utile de créer des contextes alternatifs en utilisant le constructeur de `Context`. Pour activer cet objet `Context`, utilisez la fonction *setcontext()*.

En accord avec le standard, le module *decimal* fournit des objets *Context* standards, *BasicContext* et *ExtendedContext*. Le premier est particulièrement utile pour le débogage car beaucoup des signaux ont leur déroutement activé :

```
>>> myothercontext = Context(prec=60, rounding=ROUND_HALF_DOWN)
>>> setcontext(myothercontext)
>>> Decimal(1) / Decimal(7)
Decimal('0.142857142857142857142857142857142857142857142857142857142857')

>>> ExtendedContext
Context(prec=9, rounding=ROUND_HALF_EVEN, Emin=-999999, Emax=999999,
        capitals=1, clamp=0, flags=[], traps=[])
>>> setcontext(ExtendedContext)
>>> Decimal(1) / Decimal(7)
Decimal('0.142857143')
>>> Decimal(42) / Decimal(0)
Decimal('Infinity')
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> setcontext(BasicContext)
>>> Decimal(42) / Decimal(0)
Traceback (most recent call last):
  File "<pyshell#143>", line 1, in -toplevel-
    Decimal(42) / Decimal(0)
DivisionByZero: x / 0
```

Les objets `Context` ont aussi des options pour détecter des opérations illégales lors des calculs. Ces options restent activées jusqu'à ce qu'elles soit remises à zéro de manière explicite. Il convient donc de remettre à zéro ces options avant chaque inspection de chaque calcul, avec la méthode *clear_flags()*.

```
>>> setcontext(ExtendedContext)
>>> getcontext().clear_flags()
>>> Decimal(355) / Decimal(113)
Decimal('3.14159292')
>>> getcontext()
Context(prec=9, rounding=ROUND_HALF_EVEN, Emin=-999999, Emax=999999,
        capitals=1, clamp=0, flags=[Inexact, Rounded], traps=[])
```

Les options montrent que l'approximation de π par une fraction a été arrondie (les chiffres au-delà de la précision spécifiée par l'objet *Context* ont été tronqués) et que le résultat est différent (certains des chiffres tronqués étaient différents de zéro).

L'activation du déroutement se fait en utilisant un dictionnaire dans l'attribut `traps` du contexte :

```
>>> setcontext(ExtendedContext)
>>> Decimal(1) / Decimal(0)
Decimal('Infinity')
>>> getcontext().traps[DivisionByZero] = 1
>>> Decimal(1) / Decimal(0)
Traceback (most recent call last):
  File "<pyshell#112>", line 1, in -toplevel-
    Decimal(1) / Decimal(0)
DivisionByZero: x / 0
```

La plupart des applications n'ajustent l'objet `Context` qu'une seule fois, au démarrage. Et, dans beaucoup d'applications, les données sont converties une fois pour toutes en *Decimal*. Une fois le `Context` initialisé et les objets `Decimal` créés, la majeure partie du programme manipule les données de la même manière qu'avec d'autres types numériques Python.

9.4.2 Les objets *Decimal*

class decimal.**Decimal**(*value='0'*, *context=None*)

Construit un nouvel objet *Decimal* à partir de *value*.

value peut être un entier, une chaîne de caractères, un *n*-uplet, un *float* ou une autre instance de *Decimal*. Si *value* n'est pas fourni, le constructeur renvoie `Decimal('0')`. Si *value* est une chaîne de caractères, elle doit correspondre à la syntaxe décimale en dehors des espaces de début et de fin, ou des tirets bas, qui sont enlevés :

```
sign           ::=  '+' | '-'
digit          ::=  '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
indicator      ::=  'e' | 'E'
digits         ::=  digit [digit]...
decimal-part   ::=  digits '.' [digits] | ['.'] digits
exponent-part  ::=  indicator [sign] digits
```

(suite sur la page suivante)

(suite de la page précédente)

```
infinity       ::=  'Infinity' | 'Inf'
nan            ::=  'NaN' [digits] | 'sNaN' [digits]
numeric-value  ::=  decimal-part [exponent-part] | infinity
numeric-string ::=  [sign] numeric-value | [sign] nan
```

Les chiffres codés en Unicode sont aussi autorisés, dans les emplacements `digit` ci-dessus. Cela inclut des chiffres décimaux venant d'autres alphabets (par exemple les chiffres indo-arabes ou Devanagari) ainsi que les chiffres de pleine largeur `'\uff10'` jusqu'à `'\uff19'`.

Si *value* est un *n-uplet*, il doit avoir trois éléments, le signe (`0` pour positif ou `1` pour négatif), un *n-uplet* de chiffres et un entier représentant l'exposant. Par exemple, `Decimal((0, (1, 4, 1, 4), -3))` construit l'objet `Decimal('1.414')`.

If *value* is a *float*, the binary floating-point value is losslessly converted to its exact decimal equivalent. This conversion can often require 53 or more digits of precision. For example, `Decimal(float('1.1'))` converts to `Decimal('1.1000000000000000888178419700125232338905334472656 25')`.

La précision spécifiée dans le contexte n'affecte pas le nombre de chiffres stockés. Cette valeur est déterminée exclusivement par le nombre de chiffres dans *value*. Par exemple, `Decimal('3.00000')` enregistre les 5 zéros même si la précision du contexte est de 3.

L'objectif de l'argument *context* est de déterminer ce que Python doit faire si *value* est une chaîne avec un mauvais format. Si le déroutement est activé pour *InvalidOperation*, une exception est levée, sinon le constructeur renvoie un objet `Decimal` de valeur `NaN`.

Une fois construit, un objet *Decimal* est immuable.

Modifié dans la version 3.2 : l'argument du constructeur peut désormais être un objet *float*.

Modifié dans la version 3.3 : un argument *float* lève une exception si le déroutement est activé pour *FloatOperation*. Par défaut le déroutement n'est pas activé.

Modifié dans la version 3.6 : les tirets bas sont autorisés pour regrouper, tout comme pour l'arithmétique en virgule fixe et flottante.

Decimal floating-point objects share many properties with the other built-in numeric types such as *float* and *int*. All of the usual math operations and special methods apply. Likewise, decimal objects can be copied, pickled, printed, used as dictionary keys, used as set elements, compared, sorted, and coerced to another type (such as *float* or *int*).

Il existe quelques différences mineures entre l'arithmétique entre les objets décimaux et l'arithmétique avec les entiers et les `float`. Quand l'opérateur modulo `%` est appliqué sur des objets décimaux, le signe du résultat est le signe du *dividende* plutôt que le signe du diviseur :

```
>>> (-7) % 4
1
>>> Decimal(-7) % Decimal(4)
Decimal('-3')
```

L'opérateur division entière (`//`) se comporte de la même manière, renvoyant la partie entière du quotient plutôt que son arrondi, de manière à préserver l'identité d'Euclide `x == (x // y) * y + x % y`:

```
>>> -7 // 4
-2
>>> Decimal(-7) // Decimal(4)
Decimal('-1')
```

Les opérateurs `//` et `%` implémentent la division entière et le reste (ou modulo), respectivement, tels que décrits dans la spécification.

Les objets `Decimal` ne peuvent généralement pas être combinés avec des `float` ou des objets *fractions.Fraction* lors d'opérations arithmétiques : toute addition entre un *Decimal* et un *float*, par exemple, lève une exception *TypeError*. Cependant, il est possible d'utiliser les opérateurs de comparaison entre instances

de `Decimal` et les autres types numériques. Cela évite d'avoir des résultats absurdes lors des tests d'égalité entre différents types.

Modifié dans la version 3.2 : les comparaisons inter-types entre `Decimal` et les autres types numériques sont désormais intégralement gérées.

In addition to the standard numeric properties, decimal floating-point objects also have a number of specialized methods :

adjusted()

Renvoie l'exposant ajusté après avoir décalé les chiffres les plus à droite du coefficient jusqu'à ce qu'il ne reste que le premier chiffre : `Decimal('321e+5').adjusted()` renvoie sept. Utilisée pour déterminer la position du chiffre le plus significatif par rapport à la virgule.

as_integer_ratio()

Renvoie un couple d'entiers `(n, d)` qui représentent l'instance `Decimal` donnée sous la forme d'une fraction, avec les termes les plus petits possibles et avec un dénominateur positif :

```
>>> Decimal('-3.14').as_integer_ratio()
(-157, 50)
```

La conversion est exacte. Lève une `OverflowError` sur l'infini et `ValueError` sur les *Nan*.

Ajouté dans la version 3.6.

as_tuple()

Renvoie une représentation sous la forme d'un *n-uplet nommé* du nombre `DecimalTuple(sign, digits, exposant)`.

canonical()

Renvoie la forme canonique de l'argument. Actuellement, la forme d'une instance `Decimal` est toujours canonique, donc cette opération renvoie son argument inchangé.

compare(*other*, *context=None*)

Compare les valeurs de deux instances *Decimal*. `compare()` renvoie une instance *Decimal* et, si l'un des opérandes est un *NaN*, alors le résultat est un *NaN* :

```
a or b is a NaN  ==> Decimal('NaN')
a < b            ==> Decimal('-1')
a == b           ==> Decimal('0')
a > b            ==> Decimal('1')
```

compare_signal(*other*, *context=None*)

Cette opération est identique à la méthode `compare()`, sauf que tous les *NaN* activent un déroutement. Autrement dit, si aucun des opérandes n'est un *NaN* de signalisation, alors tout opérande *NaN* silencieux est traité comme s'il s'agissait d'un *NaN* de signalisation.

compare_total(*other*, *context=None*)

Compare deux opérandes en utilisant leur représentation abstraite plutôt que leur valeur numérique. Similaire à la méthode `compare()`, mais le résultat donne un ordre total sur les instances `Decimal`. Deux instances de `Decimal` avec la même valeur numérique mais des représentations différentes se comparent de manière inégale dans cet ordre :

```
>>> Decimal('12.0').compare_total(Decimal('12'))
Decimal('-1')
```

Les *NaN* silencieux et de signalisation sont également inclus dans l'ordre total. Le résultat de cette fonction est `Decimal('0')` si les deux opérandes ont la même représentation, `Decimal('-1')` si le premier opérande est inférieur au second, et `Decimal('1')` si le premier opérande est supérieur au deuxième opérande. Voir les spécifications pour les détails de l'ordre total.

Cette opération ne dépend pas du contexte et est silencieuse : aucun indicateur n'est modifié et aucun

arrondi n'est effectué. Exceptionnellement, la version C peut lever une *InvalidOperation* si le deuxième opérande ne peut pas être converti exactement.

compare_total_mag(*other*, *context=None*)

Compare deux opérandes en utilisant leur représentation abstraite plutôt que leur valeur comme dans `compare_total()`, mais en ignorant le signe de chaque opérande. `x.compare_total_mag(y)` est équivalent à `x.copy_abs().compare_total(y.copy_abs())`.

Cette opération ne dépend pas du contexte et est silencieuse : aucun indicateur n'est modifié et aucun arrondi n'est effectué. Exceptionnellement, la version C peut lever une *InvalidOperation* si le deuxième opérande ne peut pas être converti exactement.

conjugate()

Ne fait que renvoyer self ; cette méthode existe uniquement pour se conformer à la spécification.

copy_abs()

Renvoie la valeur absolue de l'argument. Cette opération ne dépend pas du contexte et est silencieuse : aucun drapeau n'est modifié et aucun arrondi n'est effectué.

copy_negate()

Renvoie la négation de l'argument. Cette opération ne dépend pas du contexte et est silencieuse : aucun drapeau n'est modifié et aucun arrondi n'est effectué.

copy_sign(*other*, *context=None*)

Renvoie une copie du premier opérande mais avec le même signe que celui du deuxième opérande. Par exemple :

```
>>> Decimal('2.3').copy_sign(Decimal('-1.5'))
Decimal('-2.3')
```

Cette opération ne dépend pas du contexte et est silencieuse : aucun indicateur n'est modifié et aucun arrondi n'est effectué. Exceptionnellement, la version C peut lever une *InvalidOperation* si le deuxième opérande ne peut pas être converti exactement.

exp(*context=None*)

Renvoie la valeur `e**x` (fonction exponentielle) du nombre donné. Le résultat est correctement arrondi en utilisant le mode d'arrondi `ROUND_HALF_EVEN`.

```
>>> Decimal(1).exp()
Decimal('2.718281828459045235360287471')
>>> Decimal(321).exp()
Decimal('2.561702493119680037517373933E+139')
```

classmethod from_float(*f*)

Constructeur alternatif qui n'accepte que les instances de `float` ou `int`.

Remarquez que `Decimal.from_float(0.1)` est différent de `Decimal('0.1')`. Puisque 0.1 n'est pas exactement représentable en virgule flottante binaire, la valeur est stockée comme la valeur représentable la plus proche qui est `0x1.999999999999ap-4`. La valeur équivalente en décimal est `0.1000000000000000055511151231257827021181583404541015625`.

> **Note**
>
> depuis Python 3.2, une instance `Decimal` peut également être construite directement à partir d'un `float`.

```
>>> Decimal.from_float(0.1)
Decimal('0.1000000000000000055511151231257827021181583404541015625')
>>> Decimal.from_float(float('nan'))
```

(suite sur la page suivante)

(suite de la page précédente)

```
Decimal('NaN')
>>> Decimal.from_float(float('inf'))
Decimal('Infinity')
>>> Decimal.from_float(float('-inf'))
Decimal('-Infinity')
```

Ajouté dans la version 3.1.

classmethod from_number(*number*)

Alternative constructor that only accepts instances of `float`, `int` or `Decimal`, but not strings or tuples.

```
>>> Decimal.from_number(314)
Decimal('314')
>>> Decimal.from_number(0.1)
Decimal('0.1000000000000000055511151231257827021181583404541015625')
>>> Decimal.from_number(Decimal('3.14'))
Decimal('3.14')
```

Ajouté dans la version 3.14.

fma(*other*, *third*, *context=None*)

Multiplier-ajouter fusionné. Renvoie `self*other+third` sans arrondir le produit intermédiaire `self*other`.

```
>>> Decimal(2).fma(3, 5)
Decimal('11')
```

is_canonical()

Renvoie `True` si l'argument est sous forme canonique et `False` sinon. Actuellement, une instance `Decimal` est toujours canonique, donc cette opération renvoie toujours `True`.

is_finite()

Renvoie `True` si l'argument est un nombre fini et `False` si l'argument est un infini ou un *NaN*.

is_infinite()

Renvoie `True` si l'argument est un infini positif ou négatif, `False` sinon.

is_nan()

Renvoie `True` si l'argument est un *NaN* (signalétique ou silencieux), `False` sinon.

is_normal(*context=None*)

Renvoie `True` si l'argument est un nombre fini *normal*. Renvoie `False` si l'argument est zéro, infini, résultat d'un dépassement par valeur inférieure ou un *NaN*.

is_qnan()

Renvoie `True` si l'argument est un *NaN* silencieux, `False` sinon.

is_signed()

Renvoie `True` si l'argument est négatif, `False` sinon. Remarquez que les zéros et les *NaN* peuvent être signés.

is_snan()

Renvoie `True` si l'argument est un *NaN* signalétique, `False` sinon.

is_subnormal(*context=None*)

Renvoie `True` si l'argument est le résultat d'un dépassement par valeur inférieure, `False` sinon.

is_zero()

Renvoie `True` si l'argument est un zéro (positif ou négatif), `False` sinon.

ln(*context=None*)

Renvoie le logarithme naturel (base e) de l'opérande. Le résultat est arrondi avec le mode `ROUND_HALF_EVEN`.

log10(*context=None*)

Renvoie le logarithme en base 10 de l'opérande. Le résultat est arrondi avec le mode `ROUND_HALF_EVEN`.

logb(*context=None*)

Pour un nombre non nul, renvoie l'exposant ajusté de son opérande en tant qu'instance `Decimal`. Si l'opérande est un zéro alors `Decimal('-Infinity')` est renvoyé et le drapeau `DivisionByZero` est levé. Si l'opérande est un infini alors `Decimal('Infinity')` est renvoyé.

logical_and(*other*, *context=None*)

`logical_and()` est une opération logique qui prend deux *opérandes logiques* (voir *Opérandes logiques*). Le résultat est le *ET* des chiffres des deux opérandes.

logical_invert(*context=None*)

`logical_invert()` est une opération logique. Le résultat est l'inversion de chacun des chiffres de l'opérande.

logical_or(*other*, *context=None*)

`logical_or()` est une opération logique qui prend deux *opérandes logiques* (voir *Opérandes logiques*). Le résultat est le *OU* des chiffres des deux opérandes.

logical_xor(*other*, *context=None*)

`logical_xor()` est une opération logique qui prend deux *opérandes logiques* (voir *Opérandes logiques*). Le résultat est le *OU EXCLUSIF* des chiffres des deux opérandes.

max(*other*, *context=None*)

Comme `max(self, other)` sauf que la règle d'arrondi de *context* est appliquée avant le retour et que les valeurs `NaN` sont signalées ou ignorées (selon le contexte et si elles sont signalétiques ou silencieuses).

max_mag(*other*, *context=None*)

Semblable à la méthode `max()`, mais la comparaison est effectuée en utilisant les valeurs absolues des opérandes.

min(*other*, *context=None*)

Comme `min(self, other)` sauf que la règle d'arrondi de *context* est appliquée avant le retour et que les valeurs `NaN` sont signalées ou ignorées (selon le contexte et si elles sont signalétiques ou silencieuses).

min_mag(*other*, *context=None*)

Semblable à la méthode `min()`, mais la comparaison est effectuée en utilisant les valeurs absolues des opérandes.

next_minus(*context=None*)

Renvoie le plus grand nombre représentable dans le *context* donné (ou dans le contexte du fil d'exécution actuel si aucun contexte n'est donné) qui est plus petit que l'opérande donné.

next_plus(*context=None*)

Renvoie le plus petit nombre représentable dans le *context* donné (ou dans le contexte du fil d'exécution actuel si aucun contexte n'est donné) qui est supérieur à l'opérande donné.

next_toward(*other*, *context=None*)

Si les deux opérandes ne sont pas égaux, renvoie le nombre le plus proche du premier opérande dans la direction du deuxième opérande. Si les deux opérandes sont numériquement égaux, renvoie une copie du premier opérande avec le signe défini comme étant le même que le signe du second opérande.

normalize(*context=None*)

Utilisé pour produire des valeurs canoniques d'une classe d'équivalence dans le contexte actuel ou dans le contexte spécifié.

C'est la même sémantique que l'opération unaire plus, sauf que si le résultat final est fini, il est réduit à sa forme la plus simple, avec tous les zéros à droite supprimés et son signe conservé. Autrement dit,

tant que la mantisse est différente de zéro et est un multiple de dix, elle est divisée par dix et l'exposant est incrémenté de 1. Sinon (la mantisse est nulle), l'exposant est mis à 0. Dans tous les cas, le signe est inchangé.

Par exemple, `Decimal('32.100')` et `Decimal('0.321000e+2')` se normalisent tous deux à la valeur équivalente `Decimal('32.1')`.

Notez que l'arrondi est appliqué *avant* la réduction à la forme la plus simple.

Dans les dernières versions de la spécification, cette opération est également connue sous le nom de `reduce`.

number_class(*context=None*)

Renvoie une chaîne décrivant la *classe* de l'opérande. La valeur renvoyée est l'une des dix chaînes suivantes.

- `"-Infinity"`, indiquant que l'opérande est l'infini négatif ;
- `"-Normal"`, indiquant que l'opérande est un nombre négatif normal ;
- `"-Subnormal"`, indiquant que l'opérande est négatif et qu'il est le résultat d'un dépassement par valeur inférieure ;
- `"-Zero"`, indiquant que l'opérande est un zéro négatif ;
- `"+Zero"`, indiquant que l'opérande est un zéro positif ;
- `"+Subnormal"`, indiquant que l'opérande est positif et qu'il est le résultat un dépassement par valeur inférieure ;
- `"+Normal"`, indiquant que l'opérande est un nombre positif normal ;
- `"+Infinity"`, indiquant que l'opérande est l'infini positif ;
- `"NaN"`, indiquant que l'opérande est un *NaN* (*Not a Number*, pas un nombre) silencieux ;
- `"sNaN"`, indiquant que l'opérande est un *NaN* (*Not a Number*, pas un nombre) signalétique.

quantize(*exp*, *rounding=None*, *context=None*)

Renvoie une valeur égale au premier opérande après arrondi et ayant l'exposant du second opérande.

```
>>> Decimal('1.41421356').quantize(Decimal('1.000'))
Decimal('1.414')
```

Contrairement aux autres opérations, si la longueur du coefficient après l'opération de quantification est supérieure à la précision, alors une `InvalidOperation` est signalée. Ceci garantit que, sauf condition d'erreur, l'exposant quantifié est toujours égal à celui de l'opérande de droite.

Contrairement aux autres opérations, la quantification ne signale jamais de dépassement par valeur inférieure, même si le résultat est inférieur à la valeur minimale représentable et inexact.

Si l'exposant du deuxième opérande est supérieur à celui du premier, un arrondi peut être nécessaire. Dans ce cas, le mode d'arrondi est déterminé par l'argument `rounding` s'il est donné, sinon par l'argument `context` donné ; si aucun argument n'est donné, le mode d'arrondi du contexte du fil d'exécution courant est utilisé.

Une erreur est renvoyée chaque fois que l'exposant résultant est supérieur à `Emax` ou inférieur à `Etiny()`.

radix()

Renvoie `Decimal(10)`, la base (base) dans laquelle la classe `Decimal` fait toute son arithmétique. Inclus pour la compatibilité avec la spécification.

remainder_near(*other*, *context=None*)

Renvoie le reste de la division de *self* par *other*. La différence avec `self % other` réside dans le signe du reste, qui est choisi de manière à minimiser sa valeur absolue. Plus précisément, la valeur de retour est `self - n * other` où `n` est l'entier le plus proche de la valeur exacte de `self / other` et, si deux entiers sont également proches, alors l'entier pair est choisi.

Si le résultat est zéro, alors son signe est le signe de *self*.

```
>>> Decimal(18).remainder_near(Decimal(10))
Decimal('-2')
>>> Decimal(25).remainder_near(Decimal(10))
```

(suite sur la page suivante)

(suite de la page précédente)

```
Decimal('5')
>>> Decimal(35).remainder_near(Decimal(10))
Decimal('-5')
```

rotate(*other*, *context=None*)

Renvoie le résultat de la rotation des chiffres du premier opérande d'une quantité spécifiée par le deuxième opérande. Le deuxième opérande doit être un entier compris dans la plage -précision à précision. La valeur absolue du deuxième opérande donne le nombre de rotations unitaires à faire. Si le deuxième opérande est positif alors la rotation se fait vers la gauche ; sinon la rotation se fait vers la droite. Le coefficient du premier opérande est complété à gauche avec des zéros à la précision de la longueur si nécessaire. Le signe et l'exposant du premier opérande sont inchangés.

same_quantum(*other*, *context=None*)

Teste si *self* et *other* ont le même exposant ou si les deux sont `NaN`.

Cette opération ne dépend pas du contexte et est silencieuse : aucun indicateur n'est modifié et aucun arrondi n'est effectué. Exceptionnellement, la version C peut lever une *InvalidOperation* si le deuxième opérande ne peut pas être converti exactement.

scaleb(*other*, *context=None*)

Renvoie le premier opérande avec l'exposant ajusté par le second. De manière équivalente, renvoie le premier opérande multiplié par `10**other`. Le deuxième opérande doit être entier.

shift(*other*, *context=None*)

Renvoie le résultat du décalage des chiffres du premier opérande d'une quantité spécifiée par le deuxième opérande. Le deuxième opérande doit être un entier compris dans la plage -précision à précision. La valeur absolue du deuxième opérande donne le nombre de décalages unitaires à effectuer. Si le deuxième opérande est positif alors le décalage est vers la gauche ; sinon le décalage est vers la droite. Les chiffres insérés dans le nombre par le décalage sont des zéros. Le signe et l'exposant du premier opérande sont inchangés.

sqrt(*context=None*)

Renvoie la racine carrée de l'argument avec une précision maximale.

to_eng_string(*context=None*)

Convertir en chaîne, en utilisant la notation ingénieur si un exposant est nécessaire.

La notation ingénieur possède un exposant qui est un multiple de 3. Cela peut laisser jusqu'à 3 chiffres à gauche de la décimale et peut nécessiter l'ajout d'un ou de deux zéros à la fin.

Par exemple, `Decimal('123E+1')` est converti en `Decimal('1.23E+3')`.

to_integral(*rounding=None*, *context=None*)

Identique à la méthode `to_integral_value()`. Le nom `to_integral` a été conservé pour la compatibilité avec les anciennes versions.

to_integral_exact(*rounding=None*, *context=None*)

Arrondit à l'entier le plus proche, en signalant `Inexact` ou `Rounded` selon le cas si l'arrondi se produit. Le mode d'arrondi est déterminé par le paramètre `rounding` s'il est donné, sinon par le `context` donné. Si aucun paramètre n'est donné, le mode d'arrondi du contexte courant est utilisé.

to_integral_value(*rounding=None*, *context=None*)

Arrondit à l'entier le plus proche sans signaler `Inexact` ou `Rounded`. Si donné, applique *rounding* ; sinon, utilise la méthode d'arrondi dans le *context* fourni ou dans le contexte actuel.

Decimal numbers can be rounded using the `round()` function :

round(number)

round(number, ndigits)

If *ndigits* is not given or `None`, returns the nearest `int` to *number*, rounding ties to even, and igno-

ring the rounding mode of the `Decimal` context. Raises `OverflowError` if *number* is an infinity or `ValueError` if it is a (quiet or signaling) NaN.

If *ndigits* is an `int`, the context's rounding mode is respected and a `Decimal` representing *number* rounded to the nearest multiple of `Decimal('1E-ndigits')` is returned; in this case, `round(number, ndigits)` is equivalent to `self.quantize(Decimal('1E-ndigits'))`. Returns `Decimal('NaN')` if *number* is a quiet NaN. Raises `InvalidOperation` if *number* is an infinity, a signaling NaN, or if the length of the coefficient after the quantize operation would be greater than the current context's precision. In other words, for the non-corner cases :
— if *ndigits* is positive, return *number* rounded to *ndigits* decimal places;
— if *ndigits* is zero, return *number* rounded to the nearest integer;
— if *ndigits* is negative, return *number* rounded to the nearest multiple of `10**abs(ndigits)`.
Par exemple :

```
>>> from decimal import Decimal, getcontext, ROUND_DOWN
>>> getcontext().rounding = ROUND_DOWN
>>> round(Decimal('3.75'))      # context rounding ignored
4
>>> round(Decimal('3.5'))       # round-ties-to-even
4
>>> round(Decimal('3.75'), 0)   # uses the context rounding
Decimal('3')
>>> round(Decimal('3.75'), 1)
Decimal('3.7')
>>> round(Decimal('3.75'), -1)
Decimal('0E+1')
```

Opérandes logiques

Les méthodes `logical_and()`, `logical_invert()`, `logical_or()` et `logical_xor()` s'attendent à ce que leurs arguments soient des *opérandes logiques*. Un *opérande logique* est une instance `Decimal` dont l'exposant et le signe sont tous les deux zéro et dont les chiffres sont tous `0` ou `1`.

9.4.3 Objets de contexte

Les contextes sont des environnements pour les opérations arithmétiques. Ils régissent la précision, établissent des règles d'arrondi, déterminent quels signaux sont traités comme des exceptions et limitent la plage des exposants.

Chaque fil d'exécution a son propre contexte actuel qui est accessible ou modifié à l'aide des fonctions `getcontext()` et `setcontext()` :

`decimal.`**`getcontext`**`()`

Renvoie le contexte actuel du fil d'exécution courant.

`decimal.`**`setcontext`**`(`*c*`)`

Définit le contexte du fil d'exécution courant à *c*.

Vous pouvez également utiliser l'instruction `with` et la fonction `localcontext()` pour modifier temporairement le contexte actif.

`decimal.`**`localcontext`**`(`*ctx=None*, ***kwargs*`)`

Renvoie un gestionnaire de contexte qui définira le contexte actuel du fil d'exécution actif sur une copie de *ctx* à l'entrée de l'instruction *with* et restaurera le contexte précédent lors de la sortie de l'instruction *with*. Si aucun contexte n'est spécifié, une copie du contexte actuel est utilisée. L'argument *kwargs* est utilisé pour définir les attributs du nouveau contexte.

Par exemple, le code suivant définit la précision décimale actuelle à 42 chiffres, effectue un calcul, puis restaure automatiquement le contexte précédent :

```
from decimal import localcontext

with localcontext() as ctx:
    ctx.prec = 42   # Perform a high precision calculation
    s = calculate_something()
s = +s  # Round the final result back to the default precision
```

En utilisant des arguments nommés, le code serait le suivant :

```
from decimal import localcontext

with localcontext(prec=42) as ctx:
    s = calculate_something()
s = +s
```

Lève `TypeError` si *kwargs* fournit un attribut que `Context` ne prend pas en charge. Lève soit `TypeError` ou `ValueError` si *kwargs* fournit une valeur invalide pour un attribut.

Modifié dans la version 3.11 : `localcontext()` prend désormais en charge la définition des attributs de contexte grâce à l'utilisation d'arguments nommés.

De nouveaux contextes peuvent également être créés à l'aide du constructeur `Context` décrit ci-dessous. De plus, le module fournit trois contextes prédéfinis :

`decimal.`**`BasicContext`**

Il s'agit d'un contexte standard défini par la *General Decimal Arithmetic Specification*. La précision est fixée à neuf. L'arrondi est défini sur `ROUND_HALF_UP`. Tous les drapeaux sont effacés. Tous les déroutements sont activés (ils lèvent des exceptions) sauf `Inexact`, `Rounded` et `Subnormal`.

Étant donné que de nombreuses options de déroutement sont activées, ce contexte est utile pour le débogage.

`decimal.`**`ExtendedContext`**

Il s'agit d'un contexte standard défini par la *General Decimal Arithmetic Specification*. La précision est fixée à neuf. L'arrondi est défini sur `ROUND_HALF_EVEN`. Toutes les options de déroutement sont désactivées (afin que les exceptions ne soient pas levées pendant les calculs).

Comme les interruptions sont désactivées, ce contexte est utile pour les applications qui préfèrent avoir une valeur de résultat `NaN` ou `Infinity` au lieu de lever des exceptions. Cela permet à une application de terminer une exécution en présence de conditions qui, autrement, arrêteraient le programme.

`decimal.`**`DefaultContext`**

Ce contexte est utilisé par le constructeur `Context` comme prototype pour de nouveaux contextes. Changer un champ (par exemple la précision) a pour effet de changer la valeur par défaut pour les nouveaux contextes créés par le constructeur `Context`.

Ce contexte est particulièrement utile dans les environnements à plusieurs fils d'exécution. La modification de l'un des champs avant le démarrage des fils a pour effet de définir des valeurs par défaut à l'échelle du système. La modification des champs après le démarrage des fils d'exécution n'est pas recommandée car cela nécessiterait une synchronisation des fils d'exécution pour éviter des conditions de concurrence.

Dans les environnements à fil d'exécution unique, il est préférable de ne pas utiliser ce contexte du tout. Créez plutôt simplement des contextes explicitement comme décrit ci-dessous.

Les valeurs par défaut sont `Context.prec=28`, `Context.rounding=ROUND_HALF_EVEN` et les interruptions sont activées pour `Overflow`, `InvalidOperation` et `DivisionByZero`.

En plus des trois contextes fournis, de nouveaux contextes peuvent être créés avec le constructeur `Context`.

class `decimal.`**`Context`**(*prec=None*, *rounding=None*, *Emin=None*, *Emax=None*, *capitals=None*, *clamp=None*, *flags=None*, *traps=None*)

Crée un nouveau contexte. Si un champ n'est pas spécifié ou est `None`, les valeurs par défaut sont copiées à partir du `DefaultContext`. Si le champ *flags* n'est pas spécifié ou est `None`, tous les indicateurs sont effacés.

prec est un entier compris dans la plage [1, *MAX_PREC*] qui définit la précision des opérations arithmétiques dans le contexte.

L'option *rounding* est l'une des constantes répertoriées dans la section *Modes d'arrondi*.

Les champs *traps* et *flags* répertorient tous les signaux à définir. En général, les nouveaux contextes ne doivent qu'activer des surveillances et laisser les drapeaux baissés.

Les champs *Emin* et *Emax* sont des entiers spécifiant les valeurs limites autorisées pour les exposants. *Emin* doit être dans [*MIN_EMIN*, 0], *Emax* dans la plage [0, *MAX_EMAX*].

Le champ *capitals* est soit 0 soit 1 (la valeur par défaut). S'il est défini à 1, les exposants sont imprimés avec un E majuscule ; sinon, un e minuscule est utilisé : `Decimal('6.02e+23')`.

Le champ *clamp* est soit 0 (la valeur par défaut), soit 1. S'il est défini à 1, l'exposant e d'une instance *Decimal* représentable dans ce contexte est strictement limité à la plage `Emin - prec + 1 <= e <= Emax - prec + 1`. Si *clamp* est 0 alors une condition plus faible est vraie : l'exposant ajusté de l'instance *Decimal* est au plus `Emax`. Lorsque *clamp* vaut 1, un grand nombre normal voit, si possible, son exposant réduit et un nombre correspondant de zéros ajouté à sa mantisse, afin de s'adapter aux contraintes d'exposant ; cela préserve la valeur du nombre mais perd des informations sur les zéros significatifs. Par exemple :

```
>>> Context(prec=6, Emax=999, clamp=1).create_decimal('1.23e999')
Decimal('1.23000E+999')
```

Une valeur *clamp* de 1 permet la compatibilité avec les formats d'échange décimaux à largeur fixe spécifiés dans la norme IEEE 754.

La classe *Context* définit plusieurs méthodes à usage général ainsi qu'un grand nombre de méthodes permettant de faire de l'arithmétique directement dans un contexte donné. De plus, pour chacune des méthodes *Decimal* décrites ci-dessus (à l'exception des méthodes *adjusted()* et *as_tuple()*), il existe une méthode *Context* correspondante. Par exemple, pour une instance *Context* `C` et une instance *Decimal* `x`, `C.exp(x)` est équivalent à `x.exp(context=C)`. Chaque méthode *Context* accepte un entier Python (une instance de *int*) partout où une instance Decimal est acceptée.

clear_flags()

Réinitialise tous les drapeaux à 0.

clear_traps()

Réinitialise toutes les surveillances à 0.

Ajouté dans la version 3.3.

copy()

Renvoie une copie du contexte.

copy_decimal(*num*)

Renvoie une copie de l'instance `Decimal` *num*.

create_decimal(*num*)

Crée une nouvelle instance Decimal à partir de *num* mais en utilisant *self* comme contexte. Contrairement au constructeur *Decimal*, la précision du contexte, la méthode d'arrondi, les indicateurs et les surveillances sont appliqués à la conversion.

C'est utile car les constantes sont souvent données avec une précision supérieure à celle requise par l'application. Un autre avantage est que l'arrondi élimine immédiatement les effets involontaires des chiffres au-delà de la précision actuelle. Dans l'exemple suivant, l'utilisation d'entrées non arrondies signifie que l'ajout de zéro à une somme peut modifier le résultat :

```
>>> getcontext().prec = 3
>>> Decimal('3.4445') + Decimal('1.0023')
Decimal('4.45')
>>> Decimal('3.4445') + Decimal(0) + Decimal('1.0023')
Decimal('4.44')
```

Cette méthode implémente l'opération *to-number* de la spécification IBM. Si l'argument est une chaîne, aucun espace ou trait de soulignement de début ou de fin n'est autorisé.

create_decimal_from_float(*f*)

Crée une nouvelle instance Decimal à partir d'un *float f* mais en arrondissant en utilisant *self* comme contexte. Contrairement à la méthode de classe `Decimal.from_float()`, la précision du contexte, la méthode d'arrondi, les indicateurs et les surveillances sont appliqués à la conversion.

```
>>> context = Context(prec=5, rounding=ROUND_DOWN)
>>> context.create_decimal_from_float(math.pi)
Decimal('3.1415')
>>> context = Context(prec=5, traps=[Inexact])
>>> context.create_decimal_from_float(math.pi)
Traceback (most recent call last):
    ...
decimal.Inexact: None
```

Ajouté dans la version 3.1.

Etiny()

Renvoie une valeur égale à `Emin - prec + 1` qui est la valeur minimale de l'exposant pour les résultats avec dépassement inférieur. Lorsqu'un dépassement inférieur se produit, l'exposant est défini sur `Etiny`.

Etop()

Renvoie une valeur égale à `Emax - prec + 1`.

L'approche habituelle pour travailler avec des décimaux consiste à créer des instances `Decimal`, puis à appliquer des opérations arithmétiques qui ont lieu dans le contexte actuel du fil d'exécution actif. Une approche alternative consiste à utiliser des méthodes contextuelles pour calculer dans un contexte spécifique. Les méthodes sont similaires à celles de la classe `Decimal` et ne sont décrites que brièvement ici.

abs(*x*)

Renvoie la valeur absolue de *x*.

add(*x*, *y*)

Renvoie la somme de *x* et *y*.

canonical(*x*)

Renvoie l'objet Decimal *x* lui-même.

compare(*x*, *y*)

Compare *x* et *y* numériquement.

compare_signal(*x*, *y*)

Compare numériquement les valeurs des deux opérandes.

compare_total(*x*, *y*)

Compare deux opérandes en utilisant leur représentation abstraite.

compare_total_mag(*x*, *y*)

Compare deux opérandes en utilisant leur représentation abstraite, en ignorant le signe.

copy_abs(*x*)

Renvoie une copie de *x* avec le signe à 0 (c.-à-d. positif).

copy_negate(*x*)

Renvoie une copie de *x* mais de signe opposé.

copy_sign(*x*, *y*)

Copie le signe de *y* vers *x*.

divide(*x*, *y*)

Renvoie *x* divisé par *y*.

divide_int(*x*, *y*)

Renvoie *x* divisé par *y*, tronqué comme entier.

divmod(*x*, *y*)

Renvoie la partie entière de la division entre deux nombres.

exp(*x*)

Renvoie `e ** x`.

fma(*x*, *y*, *z*)

Renvoie *x* multiplié par *y*, plus *z*.

is_canonical(*x*)

Renvoie `True` si *x* est canonique ; `False` sinon.

is_finite(*x*)

Renvoie `True` si *x* est fini ; `False` sinon.

is_infinite(*x*)

Renvoie `True` si *x* est infini et `False` sinon.

is_nan(*x*)

Renvoie `True` si *x* est un *NaN* (silencieux ou signalétique), `False` sinon.

is_normal(*x*)

Renvoie `True` si *x* est un nombre normal ; `False` sinon.

is_qnan(*x*)

Renvoie `True` si *x* est un *NaN* silencieux, `False` sinon.

is_signed(*x*)

Renvoie `True` si *x* est négatif et `False` sinon.

is_snan(*x*)

Renvoie `True` si *x* est un *NaN* signalétique, `False` sinon.

is_subnormal(*x*)

Renvoie `True` si *x* est est inférieur à la valeur minimale représentable ; sinon, renvoie `False`.

is_zero(*x*)

Renvoie `True` si *x* est un zéro et `False` sinon.

ln(*x*)

Renvoie le logarithme naturel (en base e) de *x*.

log10(*x*)

Renvoie le logarithme en base 10 de *x*.

logb(*x*)

Renvoie l'exposant correspondant du chiffre de poids fort de la mantisse de l'opérande.

logical_and(*x*, *y*)

Applique l'opération logique *ET* entre les chiffres de chaque opérande.

logical_invert(*x*)

Inverse tous les chiffres de *x*.

logical_or(*x*, *y*)

Applique l'opération logique *OU* entre les chiffres de chaque opérande.

logical_xor(*x*, *y*)

Applique l'opération logique *OU EXCLUSIF* entre les chiffres de chaque opérande.

max(*x*, *y*)

Renvoie le maximum entre les deux valeurs numériques.

max_mag(*x*, *y*)

Compare les valeurs numériquement en ignorant leur signe.

min(*x*, *y*)

Compare numériquement deux valeurs et renvoie le minimum.

min_mag(*x*, *y*)

Compare les valeurs numériquement en ignorant leur signe.

minus(*x*)

Correspond à l'opérateur unaire préfixé « moins » en Python.

multiply(*x*, *y*)

Renvoie la multiplication de *x* avec *y*.

next_minus(*x*)

Renvoie le plus grand nombre représentable inférieur à *x*.

next_plus(*x*)

Renvoie le plus petit nombre représentable supérieur à *x*.

next_toward(*x*, *y*)

Renvoie le nombre le plus proche de *x*, en direction de *y*.

normalize(*x*)

Réduit *x* à sa forme la plus simple.

number_class(*x*)

Renvoie une indication de la classe de *x*.

plus(*x*)

Correspond à l'opérateur unaire préfixé « plus » en Python. Cette opération applique la précision du contexte et l'arrondi, ce n'est donc *pas* une opération d'identité.

power(*x*, *y*, *modulo=None*)

Renvoie `x` à la puissance `y`, réduit modulo `modulo` si celui-ci est donné.

Avec deux arguments, calcule `x**y`. Si `x` est négatif alors `y` doit être entier. Le résultat est inexact à moins que `y` soit entier et que le résultat soit fini et puisse être exprimé exactement en `precision` chiffres. Le mode d'arrondi du contexte est utilisé. Les résultats sont toujours correctement arrondis à la manière de Python.

`Decimal(0) ** Decimal(0)` donne `InvalidOperation` et, si `InvalidOperation` n'est pas surveillé, cela donne `Decimal('NaN')`.

Modifié dans la version 3.3 : le module C calcule *`power()`* en termes de fonctions *`exp()`* et *`ln()`* correctement arrondies. Le résultat est bien défini mais seulement « presque toujours correctement arrondi ».

Avec trois arguments, calcule `(x**y) % modulo`. Pour la forme à trois arguments, les restrictions suivantes sur les arguments s'appliquent :
— les trois arguments doivent être entiers ;
— `y` ne doit pas être négatif ;
— au moins l'un de `x` ou `y` doit être différent de zéro ;
— `modulo` doit être différent de zéro et avoir au plus `precision` chiffres.
La valeur résultant de `Context.power(x, y, modulo)` est égale à la valeur qui serait obtenue en calculant `(x**y) % modulo` avec une précision illimitée, mais est calculée plus efficacement. L'exposant du résultat est zéro, quels que soient les exposants de `x`, `y` et `modulo`. Le résultat est toujours exact.

quantize(*x*, *y*)

Renvoie une valeur égale à *x* (arrondie), ayant l'exposant de *y*.

radix()

Renvoie 10 car c'est *Decimal*, :)

remainder(*x*, *y*)

Renvoie le reste de la division entière.

Le signe du résultat, s'il est différent de zéro, est le même que celui du dividende initial.

remainder_near(*x*, *y*)

Renvoie `x - y * n`, où *n* est l'entier le plus proche de la valeur exacte de `x / y` (si le résultat est 0 alors son signe est le signe de *x*).

rotate(*x*, *y*)

Renvoie une copie pivotée de *x*, *y* fois.

same_quantum(*x*, *y*)

Renvoie `True` si les deux opérandes ont le même exposant.

scaleb(*x*, *y*)

Renvoie le premier opérande après avoir ajouté la deuxième valeur à son *exp*.

shift(*x*, *y*)

Renvoie une copie décalée de *x*, *y* fois.

sqrt(*x*)

Renvoie la racine carrée d'un nombre non négatif avec la précision donnée par le contexte.

subtract(*x*, *y*)

Renvoie la différence entre *x* et *y*.

to_eng_string(*x*)

Convertir en chaîne, en utilisant la notation ingénieur si un exposant est nécessaire.

La notation ingénieur possède un exposant qui est un multiple de 3. Cela peut laisser jusqu'à 3 chiffres à gauche de la décimale et peut nécessiter l'ajout d'un ou de deux zéros à la fin.

to_integral_exact(*x*)

Arrondit à un entier.

to_sci_string(*x*)

Convertit un nombre en chaîne en utilisant la notation scientifique.

9.4.4 Constantes

Les constantes de cette section ne sont pertinentes que pour le module C. Elles sont aussi incluses pour la compatibilité dans la version en Python pur.

	32-bit	64-bit
decimal.**MAX_PREC**	425000000	999999999999999999
decimal.**MAX_EMAX**	425000000	999999999999999999
decimal.**MIN_EMIN**	-425000000	-999999999999999999
decimal.**MIN_ETINY**	-849999999	-1999999999999999997

decimal.**HAVE_THREADS**

La valeur est `True`. Déprécié, parce que maintenant Python possède toujours des fils d'exécution.

Obsolète depuis la version 3.9.

decimal.**HAVE_CONTEXTVAR**

The default value is `True`. If Python is `configured using the --without-decimal-contextvar option`, the C version uses a thread-local rather than a coroutine-local context and the value is `False`. This is slightly faster in some nested context scenarios.

Ajouté dans la version 3.8.3.

9.4.5 Modes d'arrondi

decimal.**ROUND_CEILING**

Round towards `Infinity`.

decimal.**ROUND_DOWN**

Round towards zero.

decimal.**ROUND_FLOOR**

Round towards `-Infinity`.

decimal.**ROUND_HALF_DOWN**

Round to nearest with ties going towards zero.

decimal.**ROUND_HALF_EVEN**

Round to nearest with ties going to nearest even integer.

decimal.**ROUND_HALF_UP**

Round to nearest with ties going away from zero.

decimal.**ROUND_UP**

Round away from zero.

decimal.**ROUND_05UP**

Round away from zero if last digit after rounding towards zero would have been 0 or 5 ; otherwise round towards zero.

9.4.6 Signaux

Signals represent conditions that arise during computation. Each corresponds to one context flag and one context trap enabler.

The context flag is set whenever the condition is encountered. After the computation, flags may be checked for informational purposes (for instance, to determine whether a computation was exact). After checking the flags, be sure to clear all flags before starting the next computation.

If the context's trap enabler is set for the signal, then the condition causes a Python exception to be raised. For example, if the *DivisionByZero* trap is set, then a *DivisionByZero* exception is raised upon encountering the condition.

class decimal.**Clamped**

Altered an exponent to fit representation constraints.

Typically, clamping occurs when an exponent falls outside the context's `Emin` and `Emax` limits. If possible, the exponent is reduced to fit by adding zeros to the coefficient.

class decimal.**DecimalException**

Base class for other signals and a subclass of *ArithmeticError*.

class decimal.**DivisionByZero**

Signals the division of a non-infinite number by zero.

Can occur with division, modulo division, or when raising a number to a negative power. If this signal is not trapped, returns `Infinity` or `-Infinity` with the sign determined by the inputs to the calculation.

class decimal.**Inexact**

Indicates that rounding occurred and the result is not exact.

Signals when non-zero digits were discarded during rounding. The rounded result is returned. The signal flag or trap is used to detect when results are inexact.

class decimal.**InvalidOperation**

An invalid operation was performed.

Indicates that an operation was requested that does not make sense. If not trapped, returns `NaN`. Possible causes include :

```
Infinity - Infinity
0 * Infinity
Infinity / Infinity
x % 0
Infinity % x
sqrt(-x) and x > 0
0 ** 0
x ** (non-integer)
x ** Infinity
```

class decimal.**Overflow**

Débordement numérique.

Indicates the exponent is larger than `Context.Emax` after rounding has occurred. If not trapped, the result depends on the rounding mode, either pulling inward to the largest representable finite number or rounding outward to `Infinity`. In either case, *Inexact* and *Rounded* are also signaled.

class decimal.**Rounded**

Rounding occurred though possibly no information was lost.

Signaled whenever rounding discards digits; even if those digits are zero (such as rounding `5.00` to `5.0`). If not trapped, returns the result unchanged. This signal is used to detect loss of significant digits.

class decimal.**Subnormal**

Exponent was lower than `Emin` prior to rounding.

Occurs when an operation result is subnormal (the exponent is too small). If not trapped, returns the result unchanged.

class decimal.**Underflow**

Numerical underflow with result rounded to zero.

Occurs when a subnormal result is pushed to zero by rounding. *Inexact* and *Subnormal* are also signaled.

class decimal.**FloatOperation**

Enable stricter semantics for mixing floats and Decimals.

If the signal is not trapped (default), mixing floats and Decimals is permitted in the *Decimal* constructor, *create_decimal()* and all comparison operators. Both conversion and comparisons are exact. Any occurrence of a mixed operation is silently recorded by setting *FloatOperation* in the context flags. Explicit conversions with *from_float()* or *create_decimal_from_float()* do not set the flag.

Otherwise (the signal is trapped), only equality comparisons and explicit conversions are silent. All other mixed operations raise *FloatOperation*.

The following table summarizes the hierarchy of signals :

```
exceptions.ArithmeticError(exceptions.Exception)
    DecimalException
        Clamped
        DivisionByZero(DecimalException, exceptions.ZeroDivisionError)
        Inexact
            Overflow(Inexact, Rounded)
            Underflow(Inexact, Rounded, Subnormal)
        InvalidOperation
        Rounded
        Subnormal
        FloatOperation(DecimalException, exceptions.TypeError)
```

9.4.7 Floating-Point Notes

Mitigating round-off error with increased precision

The use of decimal floating point eliminates decimal representation error (making it possible to represent `0.1` exactly) ; however, some operations can still incur round-off error when non-zero digits exceed the fixed precision.

The effects of round-off error can be amplified by the addition or subtraction of nearly offsetting quantities resulting in loss of significance. Knuth provides two instructive examples where rounded floating-point arithmetic with insufficient precision causes the breakdown of the associative and distributive properties of addition :

```
# Examples from Seminumerical Algorithms, Section 4.2.2.
>>> from decimal import Decimal, getcontext
>>> getcontext().prec = 8

>>> u, v, w = Decimal(11111113), Decimal(-11111111), Decimal('7.51111111')
>>> (u + v) + w
Decimal('9.5111111')
>>> u + (v + w)
Decimal('10')

>>> u, v, w = Decimal(20000), Decimal(-6), Decimal('6.0000003')
>>> (u*v) + (u*w)
Decimal('0.01')
>>> u * (v+w)
Decimal('0.0060000')
```

The *decimal* module makes it possible to restore the identities by expanding the precision sufficiently to avoid loss of significance :

```
>>> getcontext().prec = 20
>>> u, v, w = Decimal(11111113), Decimal(-11111111), Decimal('7.51111111')
>>> (u + v) + w
Decimal('9.51111111')
>>> u + (v + w)
Decimal('9.51111111')
>>>
>>> u, v, w = Decimal(20000), Decimal(-6), Decimal('6.0000003')
>>> (u*v) + (u*w)
Decimal('0.0060000')
>>> u * (v+w)
Decimal('0.0060000')
```

Special values

The number system for the *decimal* module provides special values including `NaN`, `sNaN`, `-Infinity`, `Infinity`, and two zeros, `+0` and `-0`.

Infinities can be constructed directly with : `Decimal('Infinity')`. Also, they can arise from dividing by zero when the *DivisionByZero* signal is not trapped. Likewise, when the *Overflow* signal is not trapped, infinity can result from rounding beyond the limits of the largest representable number.

The infinities are signed (affine) and can be used in arithmetic operations where they get treated as very large, indeterminate numbers. For instance, adding a constant to infinity gives another infinite result.

Some operations are indeterminate and return `NaN`, or if the *InvalidOperation* signal is trapped, raise an exception. For example, `0/0` returns `NaN` which means "not a number". This variety of `NaN` is quiet and, once created, will flow through other computations always resulting in another `NaN`. This behavior can be useful for a series of computations that occasionally have missing inputs --- it allows the calculation to proceed while flagging specific results as invalid.

A variant is `sNaN` which signals rather than remaining quiet after every operation. This is a useful return value when an invalid result needs to interrupt a calculation for special handling.

The behavior of Python's comparison operators can be a little surprising where a `NaN` is involved. A test for equality where one of the operands is a quiet or signaling `NaN` always returns *False* (even when doing `Decimal('NaN')==Decimal('NaN')`), while a test for inequality always returns *True*. An attempt to compare two Decimals using any of the <, <=, > or >= operators will raise the *InvalidOperation* signal if either operand is a `NaN`, and return *False* if this signal is not trapped. Note that the General Decimal Arithmetic specification does not specify the behavior of direct comparisons; these rules for comparisons involving a `NaN` were taken from the IEEE 854 standard (see Table 3 in section 5.7). To ensure strict standards-compliance, use the *compare()* and *compare_signal()* methods instead.

The signed zeros can result from calculations that underflow. They keep the sign that would have resulted if the calculation had been carried out to greater precision. Since their magnitude is zero, both positive and negative zeros are treated as equal and their sign is informational.

In addition to the two signed zeros which are distinct yet equal, there are various representations of zero with differing precisions yet equivalent in value. This takes a bit of getting used to. For an eye accustomed to normalized floating-point representations, it is not immediately obvious that the following calculation returns a value equal to zero :

```
>>> 1 / Decimal('Infinity')
Decimal('0E-1000026')
```

9.4.8 Working with threads

The *getcontext()* function accesses a different *Context* object for each thread. Having separate thread contexts means that threads may make changes (such as `getcontext().prec=10`) without interfering with other threads.

Likewise, the *setcontext()* function automatically assigns its target to the current thread.

If *setcontext()* has not been called before *getcontext()*, then *getcontext()* will automatically create a new context for use in the current thread.

The new context is copied from a prototype context called *DefaultContext*. To control the defaults so that each thread will use the same values throughout the application, directly modify the *DefaultContext* object. This should be done *before* any threads are started so that there won't be a race condition between threads calling *getcontext()*. For example :

```
# Set applicationwide defaults for all threads about to be launched
DefaultContext.prec = 12
DefaultContext.rounding = ROUND_DOWN
DefaultContext.traps = ExtendedContext.traps.copy()
DefaultContext.traps[InvalidOperation] = 1
setcontext(DefaultContext)
```

(suite sur la page suivante)

(suite de la page précédente)

```
# Afterwards, the threads can be started
t1.start()
t2.start()
t3.start()
 . . .
```

9.4.9 Cas pratiques

Here are a few recipes that serve as utility functions and that demonstrate ways to work with the `Decimal` class :

```
def moneyfmt(value, places=2, curr='', sep=',', dp='.',
             pos='', neg='-', trailneg=''):
    """Convert Decimal to a money formatted string.

    places:  required number of places after the decimal point
    curr:    optional currency symbol before the sign (may be blank)
    sep:     optional grouping separator (comma, period, space, or blank)
    dp:      decimal point indicator (comma or period)
             only specify as blank when places is zero
    pos:     optional sign for positive numbers: '+', space or blank
    neg:     optional sign for negative numbers: '-', '(', space or blank
    trailneg:optional trailing minus indicator:  '-', ')', space or blank

    >>> d = Decimal('-1234567.8901')
    >>> moneyfmt(d, curr='$')
    '-$1,234,567.89'
    >>> moneyfmt(d, places=0, sep='.', dp='', neg='', trailneg='-')
    '1.234.568-'
    >>> moneyfmt(d, curr='$', neg='(', trailneg=')')
    '($1,234,567.89)'
    >>> moneyfmt(Decimal(123456789), sep=' ')
    '123 456 789.00'
    >>> moneyfmt(Decimal('-0.02'), neg='<', trailneg='>')
    '<0.02>'

    """
    q = Decimal(10) ** -places      # 2 places --> '0.01'
    sign, digits, exp = value.quantize(q).as_tuple()
    result = []
    digits = list(map(str, digits))
    build, next = result.append, digits.pop
    if sign:
        build(trailneg)
    for i in range(places):
        build(next() if digits else '0')
    if places:
        build(dp)
    if not digits:
        build('0')
    i = 0
    while digits:
        build(next())
        i += 1
        if i == 3 and digits:
            i = 0
            build(sep)
```

(suite sur la page suivante)

(suite de la page précédente)

```
    build(curr)
    build(neg if sign else pos)
    return ''.join(reversed(result))

def pi():
    """Compute Pi to the current precision.

    >>> print(pi())
    3.141592653589793238462643383

    """
    getcontext().prec += 2  # extra digits for intermediate steps
    three = Decimal(3)      # substitute "three=3.0" for regular floats
    lasts, t, s, n, na, d, da = 0, three, 3, 1, 0, 0, 24
    while s != lasts:
        lasts = s
        n, na = n+na, na+8
        d, da = d+da, da+32
        t = (t * n) / d
        s += t
    getcontext().prec -= 2
    return +s               # unary plus applies the new precision

def exp(x):
    """Return e raised to the power of x.  Result type matches input type.

    >>> print(exp(Decimal(1)))
    2.718281828459045235360287471
    >>> print(exp(Decimal(2)))
    7.389056098930650227230427461
    >>> print(exp(2.0))
    7.38905609893
    >>> print(exp(2+0j))
    (7.38905609893+0j)

    """
    getcontext().prec += 2
    i, lasts, s, fact, num = 0, 0, 1, 1, 1
    while s != lasts:
        lasts = s
        i += 1
        fact *= i
        num *= x
        s += num / fact
    getcontext().prec -= 2
    return +s

def cos(x):
    """Return the cosine of x as measured in radians.

    The Taylor series approximation works best for a small value of x.
    For larger values, first compute x = x % (2 * pi).

    >>> print(cos(Decimal('0.5')))
    0.8775825618903727161162815826
    >>> print(cos(0.5))
```

(suite sur la page suivante)

(suite de la page précédente)

```
    0.87758256189
    >>> print(cos(0.5+0j))
    (0.87758256189+0j)

    """
    getcontext().prec += 2
    i, lasts, s, fact, num, sign = 0, 0, 1, 1, 1, 1
    while s != lasts:
        lasts = s
        i += 2
        fact *= i * (i-1)
        num *= x * x
        sign *= -1
        s += num / fact * sign
    getcontext().prec -= 2
    return +s

def sin(x):
    """Return the sine of x as measured in radians.

    The Taylor series approximation works best for a small value of x.
    For larger values, first compute x = x % (2 * pi).

    >>> print(sin(Decimal('0.5')))
    0.4794255386042030002732879352
    >>> print(sin(0.5))
    0.479425538604
    >>> print(sin(0.5+0j))
    (0.479425538604+0j)

    """
    getcontext().prec += 2
    i, lasts, s, fact, num, sign = 1, 0, x, 1, x, 1
    while s != lasts:
        lasts = s
        i += 2
        fact *= i * (i-1)
        num *= x * x
        sign *= -1
        s += num / fact * sign
    getcontext().prec -= 2
    return +s
```

9.4.10 FAQ *decimal*

Q. C'est fastidieux de taper `decimal.Decimal('1234.5')`. Y a-t-il un moyen de réduire la frappe quand on utilise l'interpréteur interactif ?

R. Certains utilisateurs abrègent le constructeur en une seule lettre :

```
>>> D = decimal.Decimal
>>> D('1.23') + D('3.45')
Decimal('4.68')
```

Q. In a fixed-point application with two decimal places, some inputs have many places and need to be rounded. Others are not supposed to have excess digits and need to be validated. What methods should be used ?

A. The *quantize()* method rounds to a fixed number of decimal places. If the *Inexact* trap is set, it is also useful

for validation :

```
>>> TWOPLACES = Decimal(10) ** -2       # same as Decimal('0.01')
```

```
>>> # Round to two places
>>> Decimal('3.214').quantize(TWOPLACES)
Decimal('3.21')
```

```
>>> # Validate that a number does not exceed two places
>>> Decimal('3.21').quantize(TWOPLACES, context=Context(traps=[Inexact]))
Decimal('3.21')
```

```
>>> Decimal('3.214').quantize(TWOPLACES, context=Context(traps=[Inexact]))
Traceback (most recent call last):
   ...
Inexact: None
```

Q. Une fois que mes entrées sont à deux décimales valides, comment maintenir cet invariant dans l'application ?

R. Certaines opérations comme l'addition, la soustraction et la multiplication par un entier préservent automatiquement la virgule fixe. D'autres opérations, comme la division et la multiplication par des non-entiers, changent le nombre de décimales et doivent être suivies d'une étape `quantize()` :

```
>>> a = Decimal('102.72')           # Initial fixed-point values
>>> b = Decimal('3.17')
>>> a + b                           # Addition preserves fixed-point
Decimal('105.89')
>>> a - b
Decimal('99.55')
>>> a * 42                          # So does integer multiplication
Decimal('4314.24')
>>> (a * b).quantize(TWOPLACES)     # Must quantize non-integer multiplication
Decimal('325.62')
>>> (b / a).quantize(TWOPLACES)     # And quantize division
Decimal('0.03')
```

Lors du développement d'applications en virgule fixe, il est pratique de définir des fonctions pour gérer cette étape de quantification par `quantize()` :

```
>>> def mul(x, y, fp=TWOPLACES):
...     return (x * y).quantize(fp)
...
>>> def div(x, y, fp=TWOPLACES):
...     return (x / y).quantize(fp)
```

```
>>> mul(a, b)                       # Automatically preserve fixed-point
Decimal('325.62')
>>> div(b, a)
Decimal('0.03')
```

Q. There are many ways to express the same value. The numbers `200`, `200.000`, `2E2`, and `.02E+4` all have the same value at various precisions. Is there a way to transform them to a single recognizable canonical value ?

A. The `normalize()` method maps all equivalent values to a single representative :

```
>>> values = map(Decimal, '200 200.000 2E2 .02E+4'.split())
>>> [v.normalize() for v in values]
[Decimal('2E+2'), Decimal('2E+2'), Decimal('2E+2'), Decimal('2E+2')]
```

Q. When does rounding occur in a computation ?

A. It occurs *after* the computation. The philosophy of the decimal specification is that numbers are considered exact and are created independent of the current context. They can even have greater precision than current context. Computations process with those exact inputs and then rounding (or other context operations) is applied to the *result* of the computation :

```
>>> getcontext().prec = 5
>>> pi = Decimal('3.1415926535')    # More than 5 digits
>>> pi                              # All digits are retained
Decimal('3.1415926535')
>>> pi + 0                          # Rounded after an addition
Decimal('3.1416')
>>> pi - Decimal('0.00005')         # Subtract unrounded numbers, then round
Decimal('3.1415')
>>> pi + 0 - Decimal('0.00005').    # Intermediate values are rounded
Decimal('3.1416')
```

Q. Some decimal values always print with exponential notation. Is there a way to get a non-exponential representation ?

A. For some values, exponential notation is the only way to express the number of significant places in the coefficient. For example, expressing `5.0E+3` as `5000` keeps the value constant but cannot show the original's two-place significance.

If an application does not care about tracking significance, it is easy to remove the exponent and trailing zeroes, losing significance, but keeping the value unchanged :

```
>>> def remove_exponent(d):
...     return d.quantize(Decimal(1)) if d == d.to_integral() else d.normalize()
```

```
>>> remove_exponent(Decimal('5E+3'))
Decimal('5000')
```

Q. Is there a way to convert a regular float to a `Decimal` ?

A. Yes, any binary floating-point number can be exactly expressed as a Decimal though an exact conversion may take more precision than intuition would suggest :

```
>>> Decimal(math.pi)
Decimal('3.141592653589793115997963468544185161590576171875')
```

Q. Within a complex calculation, how can I make sure that I haven't gotten a spurious result because of insufficient precision or rounding anomalies.

A. The decimal module makes it easy to test results. A best practice is to re-run calculations using greater precision and with various rounding modes. Widely differing results indicate insufficient precision, rounding mode issues, ill-conditioned inputs, or a numerically unstable algorithm.

Q. I noticed that context precision is applied to the results of operations but not to the inputs. Is there anything to watch out for when mixing values of different precisions ?

A. Yes. The principle is that all values are considered to be exact and so is the arithmetic on those values. Only the results are rounded. The advantage for inputs is that "what you type is what you get". A disadvantage is that the results can look odd if you forget that the inputs haven't been rounded :

```
>>> getcontext().prec = 3
>>> Decimal('3.104') + Decimal('2.104')
Decimal('5.21')
>>> Decimal('3.104') + Decimal('0.000') + Decimal('2.104')
Decimal('5.20')
```

The solution is either to increase precision or to force rounding of inputs using the unary plus operation :

```
>>> getcontext().prec = 3
>>> +Decimal('1.23456789')       # unary plus triggers rounding
Decimal('1.23')
```

Alternatively, inputs can be rounded upon creation using the `Context.create_decimal()` method :

```
>>> Context(prec=5, rounding=ROUND_DOWN).create_decimal('1.2345678')
Decimal('1.2345')
```

Q. Is the CPython implementation fast for large numbers ?

A. Yes. In the CPython and PyPy3 implementations, the C/CFFI versions of the decimal module integrate the high speed libmpdec library for arbitrary precision correctly rounded decimal floating-point arithmetic [1]. `libmpdec` uses Karatsuba multiplication for medium-sized numbers and the Number Theoretic Transform for very large numbers.

The context must be adapted for exact arbitrary precision arithmetic. `Emin` and `Emax` should always be set to the maximum values, `clamp` should always be 0 (the default). Setting `prec` requires some care.

The easiest approach for trying out bignum arithmetic is to use the maximum value for `prec` as well [2] :

```
>>> setcontext(Context(prec=MAX_PREC, Emax=MAX_EMAX, Emin=MIN_EMIN))
>>> x = Decimal(2) ** 256
>>> x / 128
Decimal(
↪'904625697166532776746648320380374280103671755200316906558262375061821325312')
```

For inexact results, `MAX_PREC` is far too large on 64-bit platforms and the available memory will be insufficient :

```
>>> Decimal(1) / 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
MemoryError
```

On systems with overallocation (e.g. Linux), a more sophisticated approach is to adjust `prec` to the amount of available RAM. Suppose that you have 8GB of RAM and expect 10 simultaneous operands using a maximum of 500MB each :

```
>>> import sys
>>>
>>> # Maximum number of digits for a single operand using 500MB in 8-byte words
>>> # with 19 digits per word (4-byte and 9 digits for the 32-bit build):
>>> maxdigits = 19 * ((500 * 1024**2) // 8)
>>>
>>> # Check that this works:
>>> c = Context(prec=maxdigits, Emax=MAX_EMAX, Emin=MIN_EMIN)
>>> c.traps[Inexact] = True
>>> setcontext(c)
>>>
>>> # Fill the available precision with nines:
>>> x = Decimal(0).logical_invert() * 9
>>> sys.getsizeof(x)
524288112
>>> x + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  decimal.Inexact: [<class 'decimal.Inexact'>]
```

1. Ajouté dans la version 3.3.
2. Modifié dans la version 3.9 : This approach now works for all exact results except for non-integer powers.

In general (and especially on systems without overallocation), it is recommended to estimate even tighter bounds and set the `Inexact` trap if all calculations are expected to be exact.

9.5 `fractions` --- Rational numbers

Code source : Lib/fractions.py

Le module `fractions` fournit un support de l'arithmétique des nombres rationnels.

Une instance de *Fraction* peut être construite depuis une paire d'entiers, depuis un autre nombre rationnel, ou depuis une chaîne de caractères.

class fractions.**Fraction**(*numerator=0*, *denominator=1*)

class fractions.**Fraction**(*number*)

class fractions.**Fraction**(*string*)

The first version requires that *numerator* and *denominator* are instances of `numbers.Rational` and returns a new `Fraction` instance with value `numerator/denominator`. If *denominator* is `0`, it raises a `ZeroDivisionError`.

The second version requires that *number* is an instance of `numbers.Rational` or has the `as_integer_ratio()` method (this includes `float` and `decimal.Decimal`). It returns a `Fraction` instance with exactly the same value. Assumed, that the `as_integer_ratio()` method returns a pair of coprime integers and last one is positive. Note that due to the usual issues with binary point (see tut-fp-issues), the argument to `Fraction(1.1)` is not exactly equal to 11/10, and so `Fraction(1.1)` does *not* return `Fraction(11, 10)` as one might expect. (But see the documentation for the `limit_denominator()` method below.)

The last version of the constructor expects a string. The usual form for this instance is :

```
[sign] numerator ['/' denominator]
```

where the optional `sign` may be either '+' or '-' and `numerator` and `denominator` (if present) are strings of decimal digits (underscores may be used to delimit digits as with integral literals in code). In addition, any string that represents a finite value and is accepted by the `float` constructor is also accepted by the `Fraction` constructor. In either form the input string may also have leading and/or trailing whitespace. Here are some examples :

```
>>> from fractions import Fraction
>>> Fraction(16, -10)
Fraction(-8, 5)
>>> Fraction(123)
Fraction(123, 1)
>>> Fraction()
Fraction(0, 1)
>>> Fraction('3/7')
Fraction(3, 7)
>>> Fraction(' -3/7 ')
Fraction(-3, 7)
>>> Fraction('1.414213 \t\n')
Fraction(1414213, 1000000)
>>> Fraction('-.125')
Fraction(-1, 8)
>>> Fraction('7e-6')
Fraction(7, 1000000)
>>> Fraction(2.25)
Fraction(9, 4)
>>> Fraction(1.1)
```

(suite sur la page suivante)

(suite de la page précédente)

```
Fraction(2476979795053773, 2251799813685248)
>>> from decimal import Decimal
>>> Fraction(Decimal('1.1'))
Fraction(11, 10)
```

The `Fraction` class inherits from the abstract base class `numbers.Rational`, and implements all of the methods and operations from that class. `Fraction` instances are *hashable*, and should be treated as immutable. In addition, `Fraction` has the following properties and methods :

Modifié dans la version 3.2 : Le constructeur de `Fraction` accepte maintenant des instances de `float` et `decimal.Decimal`.

Modifié dans la version 3.9 : The `math.gcd()` function is now used to normalize the *numerator* and *denominator*. `math.gcd()` always returns an `int` type. Previously, the GCD type depended on *numerator* and *denominator*.

Modifié dans la version 3.11 : Underscores are now permitted when creating a `Fraction` instance from a string, following **PEP 515** rules.

Modifié dans la version 3.11 : `Fraction` implements `__int__` now to satisfy `typing.SupportsInt` instance checks.

Modifié dans la version 3.12 : Space is allowed around the slash for string inputs : `Fraction('2 / 3')`.

Modifié dans la version 3.12 : `Fraction` instances now support float-style formatting, with presentation types `"e"`, `"E"`, `"f"`, `"F"`, `"g"`, `"G"` and `"%"`.

Modifié dans la version 3.13 : Formatting of `Fraction` instances without a presentation type now supports fill, alignment, sign handling, minimum width and grouping.

Modifié dans la version 3.14 : The `Fraction` constructor now accepts any objects with the `as_integer_ratio()` method.

numerator

Numérateur de la fraction irréductible.

denominator

Dénominateur de la fraction irréductible.

as_integer_ratio()

Return a tuple of two integers, whose ratio is equal to the original Fraction. The ratio is in lowest terms and has a positive denominator.

Ajouté dans la version 3.8.

is_integer()

Return `True` if the Fraction is an integer.

Ajouté dans la version 3.12.

classmethod from_float(*flt*)

Ce constructeur alternatif accepte (uniquement) des nombres à virgule flottante, de classe `float`, ou plus généralement des instances de `numbers.Integral`. Attention, `Fraction.from_float(0.3)` est différent de `Fraction(3, 10)`.

> **Note**
>
> Depuis Python 3.2, vous pouvez aussi construire une instance de `Fraction` directement depuis un `float`.

classmethod from_decimal(*dec*)

Ce constructeur alternatif accepte (uniquement) les instances de `decimal.Decimal` ou `numbers.`

Integral.

> **Note**
>
> Depuis Python 3.2, vous pouvez aussi construire une instance de *Fraction* directement depuis une instance de *decimal.Decimal*.

classmethod from_number(*number*)

Alternative constructor which only accepts instances of *numbers.Integral*, *numbers.Rational*, *float* or *decimal.Decimal*, and objects with the `as_integer_ratio()` method, but not strings.

Ajouté dans la version 3.14.

limit_denominator(*max_denominator=1000000*)

Trouve et renvoie la *Fraction* la plus proche de `self` qui a au plus *max_denominator* comme dénominateur. Cette méthode est utile pour trouver des approximations rationnelles de nombres flottants donnés :

```
>>> from fractions import Fraction
>>> Fraction('3.1415926535897932').limit_denominator(1000)
Fraction(355, 113)
```

ou pour retrouver un nombre rationnel représenté par un flottant :

```
>>> from math import pi, cos
>>> Fraction(cos(pi/3))
Fraction(4503599627370497, 9007199254740992)
>>> Fraction(cos(pi/3)).limit_denominator()
Fraction(1, 2)
>>> Fraction(1.1).limit_denominator()
Fraction(11, 10)
```

__floor__()

Renvoie le plus grand *int* `<= self`. Cette méthode peut aussi être utilisée à travers la fonction *math.floor()* :

```
>>> from math import floor
>>> floor(Fraction(355, 113))
3
```

__ceil__()

Renvoie le plus petit *int* `>= self`. Cette méthode peut aussi être utilisée à travers la fonction *math.ceil()*.

__round__()

__round__(*ndigits*)

La première version renvoie l'*int* le plus proche de `self`, arrondissant les demis au nombre pair le plus proche. La seconde version arrondit `self` au plus proche multiple de `Fraction(1, 10**ndigits)` (logiquement, si `ndigits` est négatif), arrondissant toujours les demis au nombre pair le plus proche. Cette méthode peut aussi être utilisée à via la fonction *round()*.

__format__(*format_spec*, /)

Provides support for formatting of *Fraction* instances via the *str.format()* method, the *format()* built-in function, or Formatted string literals.

If the `format_spec` format specification string does not end with one of the presentation types `'e'`, `'E'`, `'f'`, `'F'`, `'g'`, `'G'` or `'%'` then formatting follows the general rules for fill, alignment, sign handling, minimum width, and grouping as described in the *format specification mini-language*. The "alternate

form" flag `'#'` is supported : if present, it forces the output string to always include an explicit denominator, even when the value being formatted is an exact integer. The zero-fill flag `'0'` is not supported.

If the `format_spec` format specification string ends with one of the presentation types `'e'`, `'E'`, `'f'`, `'F'`, `'g'`, `'G'` or `'%'` then formatting follows the rules outlined for the `float` type in the *Mini-langage de spécification de format* section.

Voici quelques exemples :

```
>>> from fractions import Fraction
>>> format(Fraction(103993, 33102), '_')
'103_993/33_102'
>>> format(Fraction(1, 7), '.^+10')
'...+1/7...'
>>> format(Fraction(3, 1), '')
'3'
>>> format(Fraction(3, 1), '#')
'3/1'
>>> format(Fraction(1, 7), '.40g')
'0.1428571428571428571428571428571428571429'
>>> format(Fraction('1234567.855'), '_.2f')
'1_234_567.86'
>>> f"{Fraction(355, 113):*>20.6e}"
'********3.141593e+00'
>>> old_price, new_price = 499, 672
>>> "{:.2%} price increase".format(Fraction(new_price, old_price) - 1)
'34.67% price increase'
```

Voir aussi

Module ***numbers***
Les classes abstraites représentant la hiérarchie des nombres.

9.6 `random` --- Generate pseudo-random numbers

Code source : Lib/random.py

Ce module implémente des générateurs de nombres pseudo-aléatoires pour différentes distributions.

Pour les entiers, il existe une sélection uniforme à partir d'une plage. Pour les séquences, il existe une sélection uniforme d'un élément aléatoire, une fonction pour générer une permutation aléatoire d'une liste sur place et une fonction pour un échantillonnage aléatoire sans remplacement.

Pour l'ensemble des réels, il y a des fonctions pour calculer des distributions uniformes, normales (gaussiennes), log-normales, exponentielles négatives, gamma et bêta. Pour générer des distributions d'angles, la distribution de *von Mises* est disponible.

Almost all module functions depend on the basic function `random()`, which generates a random float uniformly in the half-open range `0.0 <= X < 1.0`. Python uses the Mersenne Twister as the core generator. It produces 53-bit precision floats and has a period of 2**19937-1. The underlying implementation in C is both fast and threadsafe. The Mersenne Twister is one of the most extensively tested random number generators in existence. However, being completely deterministic, it is not suitable for all purposes, and is completely unsuitable for cryptographic purposes.

Les fonctions fournies par ce module dépendent en réalité de méthodes d'une instance cachée de la classe `random.Random`. Vous pouvez créer vos propres instances de `Random` pour obtenir des générateurs sans états partagés.

Class `Random` can also be subclassed if you want to use a different basic generator of your own devising : see the documentation on that class for more details.

Le module `random` fournit également la classe `SystemRandom` qui utilise la fonction système `os.urandom()` pour générer des nombres aléatoires à partir de sources fournies par le système d'exploitation.

Avertissement

Les générateurs pseudo-aléatoires de ce module ne doivent pas être utilisés à des fins de sécurité. Pour des utilisations de sécurité ou cryptographiques, voir le module `secrets`.

Voir aussi

M. Matsumoto and T. Nishimura, "Mersenne Twister : A 623-dimensionally equidistributed uniform pseudorandom number generator", ACM Transactions on Modeling and Computer Simulation Vol. 8, No. 1, Janvier pp.3--30 1998.

Complementary-Multiply-with-Carry recipe for a compatible alternative random number generator with a long period and comparatively simple update operations.

Note

The global random number generator and instances of `Random` are thread-safe. However, in the free-threaded build, concurrent calls to the global generator or to the same instance of `Random` may encounter contention and poor performance. Consider using separate instances of `Random` per thread instead.

9.6.1 Fonctions de gestion d'état

`random.seed(a=None, version=2)`

Initialise le générateur de nombres aléatoires.

Si *a* est omis ou `None`, l'heure système actuelle est utilisée. Si des sources aléatoires sont fournies par le système d'exploitation, elles sont utilisées à la place de l'heure système (voir la fonction `os.urandom()` pour les détails sur la disponibilité).

Si *a* est un entier, il est utilisé directement.

Avec la version 2 (par défaut), un objet `str`, `bytes` ou `bytearray` est converti en `int` et tous ses bits sont utilisés.

Avec la version 1 (fournie pour reproduire des séquences aléatoires produites par d'anciennes versions de Python), l'algorithme pour `str` et `bytes` génère une gamme plus étroite de graines.

Modifié dans la version 3.2 : Passée à la version 2 du schéma qui utilise tous les bits d'une graine de chaîne de caractères.

Modifié dans la version 3.11 : The *seed* must be one of the following types : `None`, `int`, `float`, `str`, `bytes`, or `bytearray`.

`random.getstate()`

Renvoie un objet capturant l'état interne actuel du générateur. Cet objet peut être passé à `setstate()` pour restaurer cet état.

`random.setstate(state)`

Il convient que *state* ait été obtenu à partir d'un appel précédent à `getstate()`, et `setstate()` restaure l'état interne du générateur à ce qu'il était au moment où `getstate()` a été appelé.

9.6.2 Fonctions pour les octets

random.**randbytes**(*n*)

Génère *n* octets aléatoires.

Cette méthode ne doit pas être utilisée pour générer des jetons de sécurité. Utiliser `secrets.token_bytes()` à la place.

Ajouté dans la version 3.9.

9.6.3 Fonctions pour les entiers

random.**randrange**(*stop*)

random.**randrange**(*start*, *stop*[, *step*])

Return a randomly selected element from `range(start, stop, step)`.

This is roughly equivalent to `choice(range(start, stop, step))` but supports arbitrarily large ranges and is optimized for common cases.

The positional argument pattern matches the `range()` function.

Keyword arguments should not be used because they can be interpreted in unexpected ways. For example `randrange(start=100)` is interpreted as `randrange(0, 100, 1)`.

Modifié dans la version 3.2 : `randrange()` est plus sophistiquée dans la production de valeurs uniformément distribuées. Auparavant, elle utilisait un style comme `int(random()*n)` qui pouvait produire des distributions légèrement inégales.

Modifié dans la version 3.12 : Automatic conversion of non-integer types is no longer supported. Calls such as `randrange(10.0)` and `randrange(Fraction(10, 1))` now raise a `TypeError`.

random.**randint**(*a*, *b*)

Renvoie un entier aléatoire *N* tel que `a <= N <= b`. Alias pour `randrange(a, b+1)`.

random.**getrandbits**(*k*)

Returns a non-negative Python integer with *k* random bits. This method is supplied with the Mersenne Twister generator and some other generators may also provide it as an optional part of the API. When available, `getrandbits()` enables `randrange()` to handle arbitrarily large ranges.

Modifié dans la version 3.9 : Cette méthode accepte désormais zéro pour *k*.

9.6.4 Fonctions pour les séquences

random.**choice**(*seq*)

Renvoie un élément aléatoire de la séquence non vide *seq*. Si *seq* est vide, lève `IndexError`.

random.**choices**(*population*, *weights=None*, *, *cum_weights=None*, *k=1*)

Renvoie une liste de taille *k* d'éléments choisis dans la *population* avec remise. Si la *population* est vide, lève `IndexError`.

Si une séquence de *poids* est spécifiée, les tirages sont effectués en fonction des poids relatifs. Alternativement, si une séquence *cum_weights* est donnée, les tirages sont faits en fonction des poids cumulés (peut-être calculés en utilisant `itertools.accumulate()`). Par exemple, les poids relatifs `[10, 5, 30, 5]` sont équivalents aux poids cumulatifs `[10, 15, 45, 50]`. En interne, les poids relatifs sont convertis en poids cumulatifs avant d'effectuer les tirages, ce qui vous permet d'économiser du travail en fournissant des pondérations cumulatives.

Si ni *weights* ni *cum_weights* ne sont spécifiés, les tirages sont effectués avec une probabilité uniforme. Si une séquence de poids est fournie, elle doit être de la même longueur que la séquence *population*. Spécifier à la fois *weights* et *cum_weights* lève une `TypeError`.

Les *weights* ou *cum_weights* peuvent utiliser n'importe quel type numérique interopérable avec les valeurs `float` renvoyées par `random()` (qui inclut les entiers, les flottants et les fractions mais exclut les décimaux).

Les poids sont présumés être non négatifs et finis. Une exception `ValueError` est levée si tous les poids sont à zéro.

For a given seed, the `choices()` function with equal weighting typically produces a different sequence than repeated calls to `choice()`. The algorithm used by `choices()` uses floating-point arithmetic for internal consistency and speed. The algorithm used by `choice()` defaults to integer arithmetic with repeated selections to avoid small biases from round-off error.

Ajouté dans la version 3.6.

Modifié dans la version 3.9 : Lève une `ValueError` si tous les poids sont à zéro.

`random.shuffle(x)`

Mélange la séquence *x* sans créer de nouvelle instance (« sur place »).

Pour mélanger une séquence immuable et renvoyer une nouvelle liste mélangée, utilisez `sample(x, k=len(x))` à la place.

Notez que même pour les petits `len(x)`, le nombre total de permutations de *x* peut rapidement devenir plus grand que la période de la plupart des générateurs de nombres aléatoires. Cela implique que la plupart des permutations d'une longue séquence ne peuvent jamais être générées. Par exemple, une séquence de longueur 2080 est la plus grande qui puisse tenir dans la période du générateur de nombres aléatoires Mersenne Twister.

Modifié dans la version 3.11 : Removed the optional parameter *random*.

`random.sample(population, k, *, counts=None)`

Return a *k* length list of unique elements chosen from the population sequence. Used for random sampling without replacement.

Renvoie une nouvelle liste contenant des éléments de la population tout en laissant la population originale inchangée. La liste résultante est classée par ordre de sélection de sorte que toutes les sous-tranches soient également des échantillons aléatoires valides. Cela permet aux gagnants du tirage (l'échantillon) d'être divisés en gagnants du grand prix et en gagnants de la deuxième place (les sous-tranches).

Les membres de la population n'ont pas besoin d'être *hachables* ou uniques. Si la population contient des répétitions, alors chaque occurrence est un tirage possible dans l'échantillon.

Les éléments répétés peuvent être spécifiés un à la fois ou avec le paramètre optionnel uniquement nommé *counts*. Par exemple, `sample(['red', 'blue'], counts=[4, 2], k=5)` est équivalent à `sample(['red', 'red', 'red', 'red', 'blue', 'blue'], k=5)`.

Pour choisir un échantillon parmi un intervalle d'entiers, utilisez un objet `range()` comme argument. Ceci est particulièrement rapide et économe en mémoire pour un tirage dans une grande population : `échantillon(range(10000000), k=60)`.

Si la taille de l'échantillon est supérieure à la taille de la population, une `ValueError` est levée.

Modifié dans la version 3.9 : Ajoute le paramètre *counts*.

Modifié dans la version 3.11 : The *population* must be a sequence. Automatic conversion of sets to lists is no longer supported.

9.6.5 Discrete distributions

The following function generates a discrete distribution.

`random.binomialvariate(n=1, p=0.5)`

Binomial distribution. Return the number of successes for *n* independent trials with the probability of success in each trial being *p* :

Mathematically equivalent to :

```
sum(random() < p for i in range(n))
```

The number of trials *n* should be a non-negative integer. The probability of success *p* should be between `0.0 <= p <= 1.0`. The result is an integer in the range `0 <= X <= n`.

Ajouté dans la version 3.12.

9.6.6 Distributions pour les nombres réels

Les fonctions suivantes génèrent des distributions spécifiques en nombre réels. Les paramètres de fonction sont nommés d'après les variables correspondantes de l'équation de la distribution, telles qu'elles sont utilisées dans la pratique mathématique courante ; la plupart de ces équations peuvent être trouvées dans tout document traitant de statistiques.

`random.random()`

Return the next random floating-point number in the range `0.0 <= X < 1.0`

`random.uniform(a, b)`

Return a random floating-point number *N* such that `a <= N <= b` for `a <= b` and `b <= N <= a` for `b < a`.

The end-point value `b` may or may not be included in the range depending on floating-point rounding in the expression `a + (b-a) * random()`.

`random.triangular(low, high, mode)`

Return a random floating-point number *N* such that `low <= N <= high` and with the specified *mode* between those bounds. The *low* and *high* bounds default to zero and one. The *mode* argument defaults to the midpoint between the bounds, giving a symmetric distribution.

`random.betavariate(alpha, beta)`

Distribution bêta. Les conditions sur les paramètres sont `alpha > 0` et `beta > 0`. Les valeurs renvoyées varient entre 0 et 1.

`random.expovariate(lambd=1.0)`

Distribution exponentielle. *lambd* est 1,0 divisé par la moyenne désirée. Ce ne doit pas être zéro. (Le paramètre aurait dû s'appeler "lambda", mais c'est un mot réservé en Python.) Les valeurs renvoyées vont de 0 à plus l'infini positif si *lambd* est positif, et de moins l'infini à 0 si *lambd* est négatif.

Modifié dans la version 3.12 : Added the default value for `lambd`.

`random.gammavariate(alpha, beta)`

Gamma distribution. (*Not* the gamma function !) The shape and scale parameters, *alpha* and *beta*, must have positive values. (Calling conventions vary and some sources define 'beta' as the inverse of the scale).

La fonction de distribution de probabilité est :

```
            x ** (alpha - 1) * math.exp(-x / beta)
pdf(x) =  --------------------------------------
            math.gamma(alpha) * beta ** alpha
```

`random.gauss(mu=0.0, sigma=1.0)`

Normal distribution, also called the Gaussian distribution. *mu* is the mean, and *sigma* is the standard deviation. This is slightly faster than the `normalvariate()` function defined below.

Note sur les fils d'exécution multiples (*Multithreading*) : quand deux fils d'exécution appellent cette fonction simultanément, il est possible qu'ils reçoivent la même valeur de retour. On peut l'éviter de 3 façons. 1) Avoir chaque fil utilisant une instance différente du générateur de nombres aléatoires. 2) Mettre des verrous autour de tous les appels. 3) Utiliser la fonction plus lente, mais compatible avec les programmes à fils d'exécution multiples, `normalvariate()` à la place.

Modifié dans la version 3.11 : *mu* and *sigma* now have default arguments.

`random.lognormvariate(mu, sigma)`

Logarithme de la distribution normale. Si vous prenez le logarithme naturel de cette distribution, vous obtiendrez une distribution normale avec *mu* moyen et écart-type *sigma*. *mu* peut avoir n'importe quelle valeur et *sigma* doit être supérieur à zéro.

random.**normalvariate**(*mu=0.0*, *sigma=1.0*)

Distribution normale. *mu* est la moyenne et *sigma* est l'écart type.

Modifié dans la version 3.11 : *mu* and *sigma* now have default arguments.

random.**vonmisesvariate**(*mu*, *kappa*)

mu est l'angle moyen, exprimé en radians entre 0 et 2**pi*, et *kappa* est le paramètre de concentration, qui doit être supérieur ou égal à zéro. Si *kappa* est égal à zéro, cette distribution se réduit à un angle aléatoire uniforme sur la plage de 0 à 2**pi*.

random.**paretovariate**(*alpha*)

Distribution de Pareto. *alpha* est le paramètre de forme.

random.**weibullvariate**(*alpha*, *beta*)

Distribution de Weibull. *alpha* est le paramètre de l'échelle et *beta* est le paramètre de forme.

9.6.7 Générateur alternatif

class random.**Random**([*seed*])

Classe qui implémente le générateur de nombres pseudo-aléatoires par défaut utilisé par le module `random`.

Modifié dans la version 3.11 : Formerly the *seed* could be any hashable object. Now it is limited to : `None`, `int`, `float`, `str`, `bytes`, or `bytearray`.

Subclasses of `Random` should override the following methods if they wish to make use of a different basic generator :

seed(*a=None*, *version=2*)

Override this method in subclasses to customise the `seed()` behaviour of `Random` instances.

getstate()

Override this method in subclasses to customise the `getstate()` behaviour of `Random` instances.

setstate(*state*)

Override this method in subclasses to customise the `setstate()` behaviour of `Random` instances.

random()

Override this method in subclasses to customise the `random()` behaviour of `Random` instances.

Optionally, a custom generator subclass can also supply the following method :

getrandbits(*k*)

Override this method in subclasses to customise the `getrandbits()` behaviour of `Random` instances.

class random.**SystemRandom**([*seed*])

Classe qui utilise la fonction `os.urandom()` pour générer des nombres aléatoires à partir de sources fournies par le système d'exploitation. Non disponible sur tous les systèmes. Ne repose pas sur un état purement logiciel et les séquences ne sont pas reproductibles. Par conséquent, la méthode `seed()` n'a aucun effet et est ignorée. Les méthodes `getstate()` et `setstate()` lèvent `NotImplementedError` si vous les appelez.

9.6.8 Remarques sur la reproductibilité

Sometimes it is useful to be able to reproduce the sequences given by a pseudo-random number generator. By reusing a seed value, the same sequence should be reproducible from run to run as long as multiple threads are not running.

La plupart des algorithmes et des fonctions de génération de graine du module aléatoire sont susceptibles d'être modifiés d'une version à l'autre de Python, mais deux aspects sont garantis de ne pas changer :

- Si une nouvelle méthode de génération de graine est ajoutée, une fonction rétro-compatible sera offerte.
- La méthode `random()` du générateur continuera à produire la même séquence lorsque la fonction de génération de graine compatible recevra la même semence.

9.6.9 Exemples

Exemples de base :

```
>>> random()                          # Random float:  0.0 <= x < 1.0
0.37444887175646646

>>> uniform(2.5, 10.0)                # Random float:  2.5 <= x <= 10.0
3.1800146073117523

>>> expovariate(1 / 5)                # Interval between arrivals averaging 5␣
↪seconds
5.148957571865031

>>> randrange(10)                     # Integer from 0 to 9 inclusive
7

>>> randrange(0, 101, 2)              # Even integer from 0 to 100 inclusive
26

>>> choice(['win', 'lose', 'draw'])   # Single random element from a sequence
'draw'

>>> deck = 'ace two three four'.split()
>>> shuffle(deck)                     # Shuffle a list
>>> deck
['four', 'two', 'ace', 'three']

>>> sample([10, 20, 30, 40, 50], k=4) # Four samples without replacement
[40, 10, 50, 30]
```

Simulations :

```
>>> # Six roulette wheel spins (weighted sampling with replacement)
>>> choices(['red', 'black', 'green'], [18, 18, 2], k=6)
['red', 'green', 'black', 'black', 'red', 'black']

>>> # Deal 20 cards without replacement from a deck
>>> # of 52 playing cards, and determine the proportion of cards
>>> # with a ten-value:  ten, jack, queen, or king.
>>> deal = sample(['tens', 'low cards'], counts=[16, 36], k=20)
>>> deal.count('tens') / 20
0.15

>>> # Estimate the probability of getting 5 or more heads from 7 spins
>>> # of a biased coin that settles on heads 60% of the time.
>>> sum(binomialvariate(n=7, p=0.6) >= 5 for i in range(10_000)) / 10_000
0.4169

>>> # Probability of the median of 5 samples being in middle two quartiles
>>> def trial():
...     return 2_500 <= sorted(choices(range(10_000), k=5))[2] < 7_500
...
>>> sum(trial() for i in range(10_000)) / 10_000
0.7958
```

Exemple de *bootstrapping* statistique utilisant le ré-échantillonnage avec remise pour estimer un intervalle de confiance pour la moyenne d'un échantillon :

```
# https://www.thoughtco.com/example-of-bootstrapping-3126155
from statistics import fmean as mean
from random import choices

data = [41, 50, 29, 37, 81, 30, 73, 63, 20, 35, 68, 22, 60, 31, 95]
means = sorted(mean(choices(data, k=len(data))) for i in range(100))
print(f'The sample mean of {mean(data):.1f} has a 90% confidence '
      f'interval from {means[5]:.1f} to {means[94]:.1f}')
```

Exemple d'un *resampling permutation test* pour déterminer la signification statistique ou valeur p d'une différence observée entre les effets d'un médicament et ceux d'un placebo :

```
# Example from "Statistics is Easy" by Dennis Shasha and Manda Wilson
from statistics import fmean as mean
from random import shuffle

drug = [54, 73, 53, 70, 73, 68, 52, 65, 65]
placebo = [54, 51, 58, 44, 55, 52, 42, 47, 58, 46]
observed_diff = mean(drug) - mean(placebo)

n = 10_000
count = 0
combined = drug + placebo
for i in range(n):
    shuffle(combined)
    new_diff = mean(combined[:len(drug)]) - mean(combined[len(drug):])
    count += (new_diff >= observed_diff)

print(f'{n} label reshufflings produced only {count} instances with a difference')
print(f'at least as extreme as the observed difference of {observed_diff:.1f}.')
print(f'The one-sided p-value of {count / n:.4f} leads us to reject the null')
print(f'hypothesis that there is no difference between the drug and the placebo.')
```

Simulation des heures d'arrivée et des livraisons de services pour une file d'attente de serveurs :

```
from heapq import heapify, heapreplace
from random import expovariate, gauss
from statistics import mean, quantiles

average_arrival_interval = 5.6
average_service_time = 15.0
stdev_service_time = 3.5
num_servers = 3

waits = []
arrival_time = 0.0
servers = [0.0] * num_servers  # time when each server becomes available
heapify(servers)
for i in range(1_000_000):
    arrival_time += expovariate(1.0 / average_arrival_interval)
    next_server_available = servers[0]
    wait = max(0.0, next_server_available - arrival_time)
    waits.append(wait)
    service_duration = max(0.0, gauss(average_service_time, stdev_service_time))
    service_completed = arrival_time + wait + service_duration
    heapreplace(servers, service_completed)
```

(suite sur la page suivante)

(suite de la page précédente)

```
print(f'Mean wait: {mean(waits):.1f}   Max wait: {max(waits):.1f}')
print('Quartiles:', [round(q, 1) for q in quantiles(waits)])
```

Voir aussi

Statistics for Hackers un tutoriel vidéo par Jake Vanderplas sur l'analyse statistique en utilisant seulement quelques concepts fondamentaux dont la simulation, l'échantillonnage, le brassage et la validation croisée.

Economics Simulation a simulation of a marketplace by Peter Norvig that shows effective use of many of the tools and distributions provided by this module (gauss, uniform, sample, betavariate, choice, triangular, and randrange).

A Concrete Introduction to Probability (using Python) a tutorial by Peter Norvig covering the basics of probability theory, how to write simulations, and how to perform data analysis using Python.

9.6.10 Cas pratiques

These recipes show how to efficiently make random selections from the combinatoric iterators in the `itertools` module :

```
def random_product(*args, repeat=1):
    "Random selection from itertools.product(*args, **kwds)"
    pools = [tuple(pool) for pool in args] * repeat
    return tuple(map(random.choice, pools))

def random_permutation(iterable, r=None):
    "Random selection from itertools.permutations(iterable, r)"
    pool = tuple(iterable)
    r = len(pool) if r is None else r
    return tuple(random.sample(pool, r))

def random_combination(iterable, r):
    "Random selection from itertools.combinations(iterable, r)"
    pool = tuple(iterable)
    n = len(pool)
    indices = sorted(random.sample(range(n), r))
    return tuple(pool[i] for i in indices)

def random_combination_with_replacement(iterable, r):
    "Choose r elements with replacement.  Order the result to match the iterable."
    # Result will be in set(itertools.combinations_with_replacement(iterable, r)).
    pool = tuple(iterable)
    n = len(pool)
    indices = sorted(random.choices(range(n), k=r))
    return tuple(pool[i] for i in indices)
```

Par défaut `random()` renvoie des multiples de 2^{-53} dans la plage $0.0 \le x < 1.0$. Tous ces nombres sont uniformément répartis et sont représentés exactement en tant que nombre à virgule flottante Python. Cependant, de nombreux autres nombres à virgule flottante dans cette plage, et représentables en Python, ne sont pas sélectionnables. Par exemple `0.05954861408025609` n'est pas un multiple de 2^{-53}.

La recette suivante utilise une approche différente. Tous les nombres à virgule flottante de l'intervalle sont sélectionnables. La mantisse provient d'une distribution uniforme d'entiers dans la plage $2^{52} \le mantisse < 2^{53}$. L'exposant provient d'une distribution géométrique où les exposants plus petits que *-53* apparaissent moitié moins souvent que l'exposant suivant juste plus grand.

```
from random import Random
from math import ldexp
```

(suite sur la page suivante)

(suite de la page précédente)

```
class FullRandom(Random):

    def random(self):
        mantissa = 0x10_0000_0000_0000 | self.getrandbits(52)
        exponent = -53
        x = 0
        while not x:
            x = self.getrandbits(32)
            exponent += x.bit_length() - 32
        return ldexp(mantissa, exponent)
```

Toutes les *real valued distributions* dans la classe seront utilisées dans la nouvelle méthode :

```
>>> fr = FullRandom()
>>> fr.random()
0.05954861408025609
>>> fr.expovariate(0.25)
8.87925541791544
```

La recette est conceptuellement équivalente à un algorithme qui choisit parmi tous les multiples de 2^{-1074} dans la plage $0.0 \leq x < 1.0$. Tous ces nombres sont uniformément répartis, mais la plupart doivent être arrondis au nombre à virgule Python inférieur. (La valeur 2^{-1074} est le plus petit nombre à virgule flottante positif et est égal à `math.ulp(0.0)`.)

Voir aussi

Generating Pseudo-random Floating-Point Values une publication par *Allen B. Downey* décrivant des manières de générer des nombres à virgule flottante plus fins que normalement générés par `random()`.

9.6.11 Command-line usage

Ajouté dans la version 3.13.

The `random` module can be executed from the command line.

```
python -m random [-h] [-c CHOICE [CHOICE ...] | -i N | -f N] [input ...]
```

Les options suivantes sont acceptées :

-h, --help
Affiche un message d'aide et quitte.

-c CHOICE [CHOICE ...]
--choice CHOICE [CHOICE ...]
Print a random choice, using `choice()`.

-i <N>
--integer <N>
Print a random integer between 1 and N inclusive, using `randint()`.

-f <N>
--float <N>
Print a random floating-point number between 0 and N inclusive, using `uniform()`.

If no options are given, the output depends on the input :
- String or multiple : same as `--choice`.
- Integer : same as `--integer`.
- Float : same as `--float`.

9.6.12 Command-line example

Here are some examples of the `random` command-line interface :

```
$ # Choose one at random
$ python -m random egg bacon sausage spam "Lobster Thermidor aux crevettes with a␣
↪Mornay sauce"
Lobster Thermidor aux crevettes with a Mornay sauce

$ # Random integer
$ python -m random 6
6

$ # Random floating-point number
$ python -m random 1.8
1.7080016272295635

$ # With explicit arguments
$ python  -m random --choice egg bacon sausage spam "Lobster Thermidor aux␣
↪crevettes with a Mornay sauce"
egg

$ python -m random --integer 6
3

$ python -m random --float 1.8
1.5666339105010318

$ python -m random --integer 6
5

$ python -m random --float 6
3.1942323316565915
```

9.7 `statistics` --- Mathematical statistics functions

Ajouté dans la version 3.4.

Code source : Lib/statistics.py

Ce module fournit des fonctions pour le calcul de valeurs statistiques sur des données numériques (valeurs réelles de la classe *`Real`*).

The module is not intended to be a competitor to third-party libraries such as NumPy, SciPy, or proprietary full-featured statistics packages aimed at professional statisticians such as Minitab, SAS and Matlab. It is aimed at the level of graphing and scientific calculators.

À moins que cela ne soit précisé différemment, ces fonctions gèrent les objets *`int`*, *`float`*, *`Decimal`* et *`Fraction`*. Leur bon comportement avec d'autres types (numériques ou non) n'est pas garanti. Le comportement de ces fonctions sur des collections mixtes de différents types est indéfini et dépend de l'implémentation. Si vos données comportement un mélange de plusieurs types, vous pouvez utiliser *`map()`* pour vous assurer que le résultat est cohérent, par exemple : `map(float, input_data)`.

Some datasets use `NaN` (not a number) values to represent missing data. Since NaNs have unusual comparison semantics, they cause surprising or undefined behaviors in the statistics functions that sort data or that count occurrences. The functions affected are `median()`, `median_low()`, `median_high()`, `median_grouped()`, `mode()`, `multimode()`, and `quantiles()`. The `NaN` values should be stripped before calling these functions :

```
>>> from statistics import median
>>> from math import isnan
>>> from itertools import filterfalse

>>> data = [20.7, float('NaN'),19.2, 18.3, float('NaN'), 14.4]
>>> sorted(data)  # This has surprising behavior
[20.7, nan, 14.4, 18.3, 19.2, nan]
>>> median(data)  # This result is unexpected
16.35

>>> sum(map(isnan, data))     # Number of missing values
2
>>> clean = list(filterfalse(isnan, data))  # Strip NaN values
>>> clean
[20.7, 19.2, 18.3, 14.4]
>>> sorted(clean)  # Sorting now works as expected
[14.4, 18.3, 19.2, 20.7]
>>> median(clean)       # This result is now well defined
18.75
```

9.7.1 Moyennes et mesures de la tendance centrale

Ces fonctions calculent une moyenne ou une valeur typique à partir d'une population ou d'un échantillon.

`mean()`	Moyenne arithmétique des données.
`fmean()`	Fast, floating-point arithmetic mean, with optional weighting.
`geometric_mean()`	Moyenne géométrique des données.
`harmonic_mean()`	Moyenne harmonique des données.
`kde()`	Estimate the probability density distribution of the data.
`kde_random()`	Random sampling from the PDF generated by kde().
`median()`	Médiane (valeur centrale) des données.
`median_low()`	Médiane basse des données.
`median_high()`	Médiane haute des données.
`median_grouped()`	Median (50th percentile) of grouped data.
`mode()`	Mode principal (la valeur la plus représentée) de données discrètes ou nominales.
`multimode()`	Liste des modes (les valeurs les plus représentées) de données discrètes ou nominales.
`quantiles()`	Divise les données en intervalles de même probabilité.

9.7.2 Mesures de la dispersion

Ces fonctions mesurent la tendance de la population ou d'un échantillon à dévier des valeurs typiques ou des valeurs moyennes.

`pstdev()`	Écart-type de la population.
`pvariance()`	Variance de la population.
`stdev()`	Écart-type d'un échantillon.
`variance()`	Variance d'un échantillon.

9.7.3 Statistics for relations between two inputs

These functions calculate statistics regarding relations between two inputs.

`covariance()`	Covariance d'un échantillon pour deux variables.
`correlation()`	Pearson and Spearman's correlation coefficients.
`linear_regression()`	Slope and intercept for simple linear regression.

9.7.4 Détails des fonctions

Note : les fonctions ne requièrent pas que les données soient ordonnées. Toutefois, pour en faciliter la lecture, les exemples utiliseront des séquences croissantes.

`statistics.mean(data)`

Renvoie la moyenne arithmétique de l'échantillon *data* qui peut être une séquence ou un itérable.

La moyenne arithmétique est la somme des valeurs divisée par le nombre d'observations. Il s'agit de la valeur couramment désignée comme la « moyenne » bien qu'il existe de multiples façons de définir mathématiquement la moyenne. C'est une mesure de la tendance centrale des données.

Une erreur `StatisticsError` est levée si *data* est vide.

Exemples d'utilisation :

```
>>> mean([1, 2, 3, 4, 4])
2.8
>>> mean([-1.0, 2.5, 3.25, 5.75])
2.625

>>> from fractions import Fraction as F
>>> mean([F(3, 7), F(1, 21), F(5, 3), F(1, 3)])
Fraction(13, 21)

>>> from decimal import Decimal as D
>>> mean([D("0.5"), D("0.75"), D("0.625"), D("0.375")])
Decimal('0.5625')
```

Note

La moyenne arithmétique est fortement impactée par la présence de valeurs aberrantes et n'est pas un estimateur robuste de la tendance centrale : la moyenne n'est pas nécessairement un exemple représentatif de l'échantillon. Voir `median()` pour des mesures plus robustes de la tendance centrale.

La moyenne de l'échantillon est une estimation non biaisée de la moyenne de la véritable population. Ainsi, en calculant la moyenne sur tous les échantillons possibles, `mean(sample)` converge vers la moyenne réelle de la population entière. Si *data* est une population entière plutôt qu'un échantillon, alors `mean(data)` équivaut à calculer la véritable moyenne μ.

`statistics.fmean(data, weights=None)`

Convertit *data* en nombres à virgule flottante et calcule la moyenne arithmétique.

Cette fonction est plus rapide que `mean()` et renvoie toujours un `float`. *data* peut être une séquence ou un itérable. Si les données d'entrée sont vides, la fonction lève une erreur `StatisticsError`.

```
>>> fmean([3.5, 4.0, 5.25])
4.25
```

Optional weighting is supported. For example, a professor assigns a grade for a course by weighting quizzes at 20%, homework at 20%, a midterm exam at 30%, and a final exam at 30% :

```
>>> grades = [85, 92, 83, 91]
>>> weights = [0.20, 0.20, 0.30, 0.30]
>>> fmean(grades, weights)
87.6
```

If *weights* is supplied, it must be the same length as the *data* or a `ValueError` will be raised.

Ajouté dans la version 3.8.

Modifié dans la version 3.11 : Added support for *weights*.

`statistics.geometric_mean(data)`

Convertit *data* en nombres à virgule flottante et calcule la moyenne géométrique.

La moyenne géométrique mesure la tendance centrale ou la valeur typique de *data* en utilisant le produit des valeurs (par opposition à la moyenne arithmétique qui utilise la somme).

Lève une erreur `StatisticsError` si les données sont vides, si elles contiennent un zéro ou une valeur négative. *data* peut être une séquence ou un itérable.

Aucune mesure particulière n'est prise pour garantir que le résultat est parfaitement exact (cela peut toutefois changer dans une version future).

```
>>> round(geometric_mean([54, 24, 36]), 1)
36.0
```

Ajouté dans la version 3.8.

`statistics.harmonic_mean(data, weights=None)`

Return the harmonic mean of *data*, a sequence or iterable of real-valued numbers. If *weights* is omitted or `None`, then equal weighting is assumed.

The harmonic mean is the reciprocal of the arithmetic `mean()` of the reciprocals of the data. For example, the harmonic mean of three values *a*, *b* and *c* will be equivalent to `3/(1/a + 1/b + 1/c)`. If one of the values is zero, the result will be zero.

The harmonic mean is a type of average, a measure of the central location of the data. It is often appropriate when averaging ratios or rates, for example speeds.

Supposons qu'une voiture parcoure 10 km à 40 km/h puis 10 km à 60 km/h. Quelle a été sa vitesse moyenne ?

```
>>> harmonic_mean([40, 60])
48.0
```

Suppose a car travels 40 km/hr for 5 km, and when traffic clears, speeds-up to 60 km/hr for the remaining 30 km of the journey. What is the average speed ?

```
>>> harmonic_mean([40, 60], weights=[5, 30])
56.0
```

`StatisticsError` is raised if *data* is empty, any element is less than zero, or if the weighted sum isn't positive.

L'algorithme actuellement implémenté s'arrête prématurément lors de la rencontre d'un zéro dans le parcours de l'entrée. Cela signifie que la validité des valeurs suivantes n'est pas testée (ce comportement est susceptible de changer dans une version future).

Ajouté dans la version 3.6.

Modifié dans la version 3.10 : Added support for *weights*.

`statistics.kde(data, h, kernel='normal', *, cumulative=False)`

Kernel Density Estimation (KDE) : Create a continuous probability density function or cumulative distribution function from discrete samples.

The basic idea is to smooth the data using a kernel function. to help draw inferences about a population from a sample.

The degree of smoothing is controlled by the scaling parameter *h* which is called the bandwidth. Smaller values emphasize local features while larger values give smoother results.

The *kernel* determines the relative weights of the sample data points. Generally, the choice of kernel shape does not matter as much as the more influential bandwidth smoothing parameter.

Kernels that give some weight to every sample point include *normal* (*gauss*), *logistic*, and *sigmoid*.

Kernels that only give weight to sample points within the bandwidth include *rectangular* (*uniform*), *triangular*, *parabolic* (*epanechnikov*), *quartic* (*biweight*), *triweight*, and *cosine*.

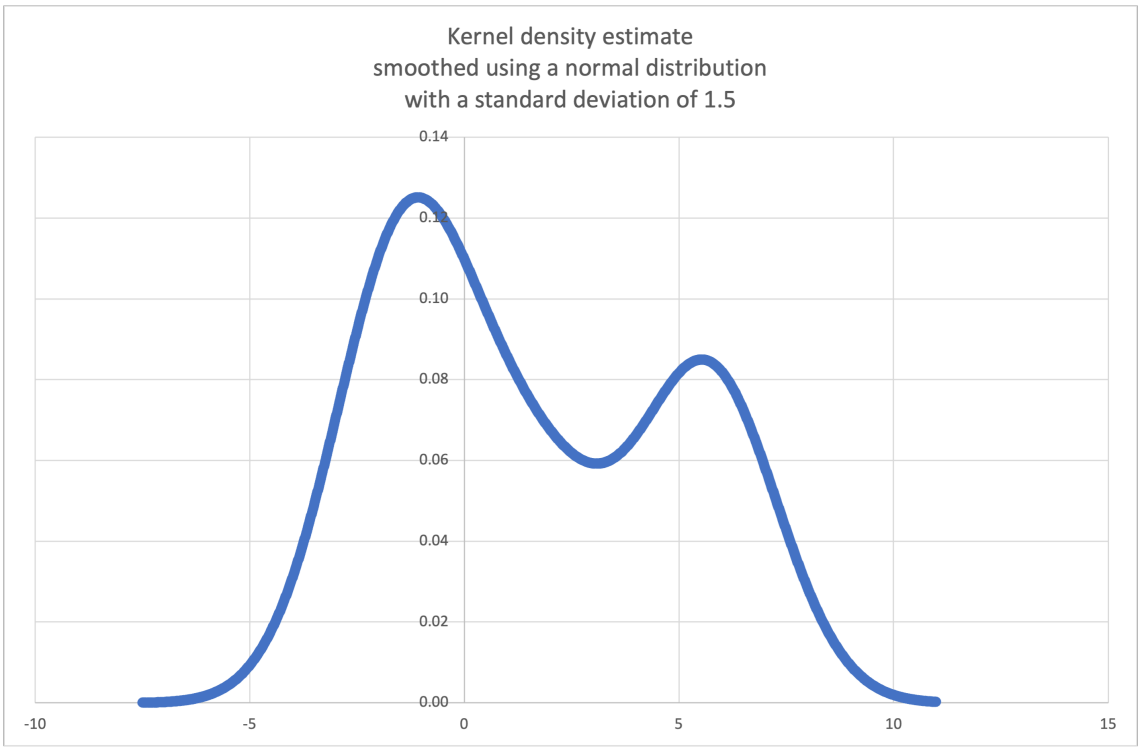
If *cumulative* is true, will return a cumulative distribution function.

A `StatisticsError` will be raised if the *data* sequence is empty.

Wikipedia has an example where we can use `kde()` to generate and plot a probability density function estimated from a small sample :

```
>>> sample = [-2.1, -1.3, -0.4, 1.9, 5.1, 6.2]
>>> f_hat = kde(sample, h=1.5)
>>> xarr = [i/100 for i in range(-750, 1100)]
>>> yarr = [f_hat(x) for x in xarr]
```

The points in `xarr` and `yarr` can be used to make a PDF plot :



Ajouté dans la version 3.13.

`statistics.kde_random`(*data*, *h*, *kernel='normal'*, ***, *seed=None*)

Return a function that makes a random selection from the estimated probability density function produced by `kde(data, h, kernel)`.

Providing a *seed* allows reproducible selections. In the future, the values may change slightly as more accurate kernel inverse CDF estimates are implemented. The seed may be an integer, float, str, or bytes.

A `StatisticsError` will be raised if the *data* sequence is empty.

Continuing the example for `kde()`, we can use `kde_random()` to generate new random selections from an estimated probability density function :

```
>>> data = [-2.1, -1.3, -0.4, 1.9, 5.1, 6.2]
>>> rand = kde_random(data, h=1.5, seed=8675309)
>>> new_selections = [rand() for i in range(10)]
>>> [round(x, 1) for x in new_selections]
[0.7, 6.2, 1.2, 6.9, 7.0, 1.8, 2.5, -0.5, -1.8, 5.6]
```

Ajouté dans la version 3.13.

statistics.**median**(*data*)

Renvoie la médiane (la valeur centrale) de données numériques en utilisant la méthode classique « moyenne des deux du milieu ». Lève une erreur `StatisticsError` si *data* est vide. *data* peut être une séquence ou un itérable.

La médiane est une mesure robuste de la tendance centrale et est moins sensible à la présence de valeurs aberrantes que la moyenne. Lorsque le nombre d'observations est impair, la valeur du milieu est renvoyée :

```
>>> median([1, 3, 5])
3
```

Lorsque le nombre d'observations est pair, la médiane est interpolée en calculant la moyenne des deux valeurs du milieu :

```
>>> median([1, 3, 5, 7])
4.0
```

Cette approche est adaptée à des données discrètes à condition que vous acceptiez que la médiane ne fasse pas nécessairement partie des observations.

Si les données sont ordinales (elles peuvent être ordonnées) mais pas numériques (elles ne peuvent être additionnées), utilisez `median_low()` ou `median_high()` à la place.

statistics.**median_low**(*data*)

Renvoie la médiane basse de données numériques. Lève une erreur `StatisticsError` si *data* est vide. *data* peut être une séquence ou un itérable.

La médiane basse est toujours une valeur représentée dans les données. Lorsque le nombre d'observations est impair, la valeur du milieu est renvoyée. Sinon, la plus petite des deux valeurs du milieu est renvoyée.

```
>>> median_low([1, 3, 5])
3
>>> median_low([1, 3, 5, 7])
3
```

Utilisez la médiane basse lorsque vos données sont discrètes et que vous préférez que la médiane soit une valeur représentée dans vos observations plutôt que le résultat d'une interpolation.

statistics.**median_high**(*data*)

Renvoie la médiane haute des données. Lève une erreur `StatisticsError` si *data* est vide. *data* peut être une séquence ou un itérable.

La médiane haute est toujours une valeur représentée dans les données. Lorsque le nombre d'observations est impair, la valeur du milieu est renvoyée. Sinon, la plus grande des deux valeurs du milieu est renvoyée.

```
>>> median_high([1, 3, 5])
3
>>> median_high([1, 3, 5, 7])
5
```

Utilisez la médiane haute lorsque vos données sont discrètes et que vous préférez que la médiane soit une valeur représentée dans vos observations plutôt que le résultat d'une interpolation.

statistics.**median_grouped**(*data*, *interval=1.0*)

Estimates the median for numeric data that has been grouped or binned around the midpoints of consecutive, fixed-width intervals.

The *data* can be any iterable of numeric data with each value being exactly the midpoint of a bin. At least one value must be present.

The *interval* is the width of each bin.

For example, demographic information may have been summarized into consecutive ten-year age groups with

each group being represented by the 5-year midpoints of the intervals :

```
>>> from collections import Counter
>>> demographics = Counter({
...    25: 172,   # 20 to 30 years old
...    35: 484,   # 30 to 40 years old
...    45: 387,   # 40 to 50 years old
...    55:  22,   # 50 to 60 years old
...    65:   6,   # 60 to 70 years old
... })
...
```

The 50th percentile (median) is the 536th person out of the 1071 member cohort. That person is in the 30 to 40 year old age group.

The regular *median()* function would assume that everyone in the tricenarian age group was exactly 35 years old. A more tenable assumption is that the 484 members of that age group are evenly distributed between 30 and 40. For that, we use *median_grouped()* :

```
>>> data = list(demographics.elements())
>>> median(data)
35
>>> round(median_grouped(data, interval=10), 1)
37.5
```

The caller is responsible for making sure the data points are separated by exact multiples of *interval*. This is essential for getting a correct result. The function does not check this precondition.

Inputs may be any numeric type that can be coerced to a float during the interpolation step.

statistics.**mode**(*data*)

Renvoie la valeur la plus représentée dans la collection *data* (discrète ou nominale). Ce mode, lorsqu'il existe, est la valeur la plus représentative des données et est une mesure de la tendance centrale.

S'il existe plusieurs modes avec la même fréquence, cette fonction renvoie le premier qui a été rencontré dans *data*. Utilisez `min(multimode(data))` ou `max(multimode(data))` si vous désirez le plus petit mode ou le plus grand mode. Lève une erreur *StatisticsError* si *data* est vide.

`mode` suppose que les données sont discrètes et renvoie une seule valeur. Il s'agit de la définition usuelle du mode telle qu'enseignée dans à l'école :

```
>>> mode([1, 1, 2, 3, 3, 3, 3, 4])
3
```

Le mode a la particularité d'être la seule statistique de ce module à pouvoir être calculée sur des données nominales (non numériques) :

```
>>> mode(["red", "blue", "blue", "red", "green", "red", "red"])
'red'
```

Only hashable inputs are supported. To handle type *set*, consider casting to *frozenset*. To handle type *list*, consider casting to *tuple*. For mixed or nested inputs, consider using this slower quadratic algorithm that only depends on equality tests : `max(data, key=data.count)`.

Modifié dans la version 3.8 : Gère désormais des jeux de données avec plusieurs modes en renvoyant le premier mode rencontré. Précédemment, une erreur *StatisticsError* était levée si plusieurs modes étaient trouvés.

statistics.**multimode**(*data*)

Renvoie une liste des valeurs les plus fréquentes en suivant leur ordre d'apparition dans *data*. Renvoie plusieurs résultats s'il y a plusieurs modes ou une liste vide si *data* est vide :

```
>>> multimode('aabbbbccddddeeffffgg')
['b', 'd', 'f']
>>> multimode('')
[]
```

Ajouté dans la version 3.8.

statistics.**pstdev**(*data*, *mu=None*)

Renvoie l'écart-type de la population (racine carrée de la variance de la population). Voir `pvariance()` pour les arguments et d'autres précisions.

```
>>> pstdev([1.5, 2.5, 2.5, 2.75, 3.25, 4.75])
0.986893273527251
```

statistics.**pvariance**(*data*, *mu=None*)

Renvoie la variance de la population *data*, une séquence non-vide ou un itérable de valeurs réelles. La variance, ou moment de second ordre, est une mesure de la variabilité (ou dispersion) des données. Une variance élevée indique une large dispersion des valeurs ; une faible variance indique que les valeurs sont resserrées autour de la moyenne.

If the optional second argument *mu* is given, it should be the *population* mean of the *data*. It can also be used to compute the second moment around a point that is not the mean. If it is missing or `None` (the default), the arithmetic mean is automatically calculated.

Utilisez cette fonction pour calculer la variance sur une population complète. Pour estimer la variance à partir d'un échantillon, utilisez plutôt `variance()` à la place.

Lève une erreur `StatisticsError` si *data* est vide.

Exemples :

```
>>> data = [0.0, 0.25, 0.25, 1.25, 1.5, 1.75, 2.75, 3.25]
>>> pvariance(data)
1.25
```

Si vous connaissez la moyenne de vos données, il est possible de la passer comme argument optionnel *mu* lors de l'appel de fonction pour éviter de la calculer une nouvelle fois :

```
>>> mu = mean(data)
>>> pvariance(data, mu)
1.25
```

La fonction gère les nombres décimaux et les fractions :

```
>>> from decimal import Decimal as D
>>> pvariance([D("27.5"), D("30.25"), D("30.25"), D("34.5"), D("41.75")])
Decimal('24.815')

>>> from fractions import Fraction as F
>>> pvariance([F(1, 4), F(5, 4), F(1, 2)])
Fraction(13, 72)
```

Note

Cette fonction renvoie la variance de la population σ^2 lorsqu'elle est appliquée sur la population entière. Si elle est appliquée seulement sur un échantillon, le résultat est alors la variance de l'échantillon s^2 ou variance à N degrés de liberté.

Si vous connaissez d'avance la vraie moyenne de la population μ, vous pouvez utiliser cette fonction pour calculer la variance de l'échantillon sachant la moyenne de la population en la passante comme second

argument. En supposant que les observations sont issues d'un tirage aléatoire uniforme dans la population, le résultat sera une estimation non biaisée de la variance de la population.

statistics.**stdev**(*data*, *xbar=None*)

Renvoie l'écart-type de l'échantillon (racine carrée de la variance de l'échantillon). Voir `variance()` pour les arguments et plus de détails.

```
>>> stdev([1.5, 2.5, 2.5, 2.75, 3.25, 4.75])
1.0810874155219827
```

statistics.**variance**(*data*, *xbar=None*)

Renvoie la variance de l'échantillon *data*, un itérable d'au moins deux valeurs réelles. La variance, ou moment de second ordre, est une mesure de la variabilité (ou dispersion) des données. Une variance élevée indique que les données sont très dispersées ; une variance faible indique que les valeurs sont resserrées autour de la moyenne.

If the optional second argument *xbar* is given, it should be the *sample* mean of *data*. If it is missing or `None` (the default), the mean is automatically calculated.

Utilisez cette fonction lorsque vos données forment un échantillon d'une population plus grande. Pour calculer la variance d'une population complète, utilisez `pvariance()`.

Lève une erreur `StatisticsError` si *data* contient moins de deux éléments.

Exemples :

```
>>> data = [2.75, 1.75, 1.25, 0.25, 0.5, 1.25, 3.5]
>>> variance(data)
1.3720238095238095
```

If you have already calculated the sample mean of your data, you can pass it as the optional second argument *xbar* to avoid recalculation :

```
>>> m = mean(data)
>>> variance(data, m)
1.3720238095238095
```

Cette fonction ne vérifie pas que la valeur passée dans l'argument *xbar* correspond bien à la moyenne. Utiliser des valeurs arbitraires pour *xbar* produit des résultats impossibles ou incorrects.

La fonction gère les nombres décimaux et les fractions :

```
>>> from decimal import Decimal as D
>>> variance([D("27.5"), D("30.25"), D("30.25"), D("34.5"), D("41.75")])
Decimal('31.01875')

>>> from fractions import Fraction as F
>>> variance([F(1, 6), F(1, 2), F(5, 3)])
Fraction(67, 108)
```

Note

Cela correspond à la variance s^2 de l'échantillon avec correction de Bessel (ou variance à N-1 degrés de liberté). En supposant que les observations sont représentatives de la population (c'est-à-dire indépendantes et identiquement distribuées), alors le résultat est une estimation non biaisée de la variance.

Si vous connaissez d'avance la vraie moyenne μ de la population, vous devriez la passer à `pvariance()` comme paramètre *mu* pour obtenir la variance de l'échantillon.

statistics.**quantiles**(*data*, *, *n=4*, *method='exclusive'*)

Divise *data* en *n* intervalles réels de même probabilité. Renvoie une liste de `n - 1` valeurs délimitant les intervalles (les quantiles).

Utilisez `n = 4` pour obtenir les quartiles (le défaut), `n = 10` pour obtenir les déciles et `n = 100` pour obtenir les centiles (ce qui produit 99 valeurs qui séparent *data* en 100 groupes de même taille). Lève une erreur `StatisticsError` si *n* est strictement inférieur à 1.

The *data* can be any iterable containing sample data. For meaningful results, the number of data points in *data* should be larger than *n*. Raises `StatisticsError` if there is not at least one data point.

Les quantiles sont linéairement interpolées à partir des deux valeurs les plus proches dans l'échantillon. Par exemple, si un quantile se situe à un tiers de la distance entre les deux valeurs de l'échantillon `100` et `112`, le quantile vaudra `104`.

L'argument *method* indique la méthode à utiliser pour calculer les quantiles et peut être modifié pour spécifier s'il faut inclure ou exclure les valeurs basses et hautes de *data* de la population.

La valeur par défaut pour *method* est `"exclusive"` et est applicable pour des données échantillonnées dans une population dont une des valeurs extrêmes peut être plus grande (respectivement plus petite) que le maximum (respectivement le minimum) des valeurs de l'échantillon. La proportion de la population se situant en-dessous du i^{e} de *m* valeurs ordonnées est calculée par la formule `i / (m + 1)`. Par exemple, en supposant 9 valeurs dans l'échantillon, cette méthode les ordonne et leur associe les quantiles suivants : 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, 90%.

En utilisant `"inclusive"` comme valeur du paramètre *method*, on suppose que *data* correspond soit à une population entière, soit que les valeurs extrêmes de la population sont représentées dans l'échantillon. Le minimum de *data* est alors considéré comme 0e centile et le maximum comme 100e centile. La proportion de la population se situant sous la i^{e} valeur de *m* valeurs ordonnées est calculée par la formule `(i - 1)/(m - 1)`. En supposant que l'on a 11 valeurs dans l'échantillon, cette méthode les ordonne et leur associe les quantiles suivants : 0%, 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, 90%, 100%.

```
# Decile cut points for empirically sampled data
>>> data = [105, 129, 87, 86, 111, 111, 89, 81, 108, 92, 110,
...         100, 75, 105, 103, 109, 76, 119, 99, 91, 103, 129,
...         106, 101, 84, 111, 74, 87, 86, 103, 103, 106, 86,
...         111, 75, 87, 102, 121, 111, 88, 89, 101, 106, 95,
...         103, 107, 101, 81, 109, 104]
>>> [round(q, 1) for q in quantiles(data, n=10)]
[81.0, 86.2, 89.0, 99.4, 102.5, 103.6, 106.0, 109.8, 111.0]
```

Ajouté dans la version 3.8.

Modifié dans la version 3.13 : No longer raises an exception for an input with only a single data point. This allows quantile estimates to be built up one sample point at a time becoming gradually more refined with each new data point.

statistics.**covariance**(*x*, *y*, /)

Return the sample covariance of two inputs *x* and *y*. Covariance is a measure of the joint variability of two inputs.

Both inputs must be of the same length (no less than two), otherwise `StatisticsError` is raised.

Exemples :

```
>>> x = [1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> y = [1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> covariance(x, y)
0.75
>>> z = [9, 8, 7, 6, 5, 4, 3, 2, 1]
>>> covariance(x, z)
-7.5
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> covariance(z, x)
-7.5
```

Ajouté dans la version 3.10.

`statistics.correlation`(*x*, *y*, /, *, *method='linear'*)

Return the Pearson's correlation coefficient for two inputs. Pearson's correlation coefficient *r* takes values between -1 and +1. It measures the strength and direction of a linear relationship.

If *method* is "ranked", computes Spearman's rank correlation coefficient for two inputs. The data is replaced by ranks. Ties are averaged so that equal values receive the same rank. The resulting coefficient measures the strength of a monotonic relationship.

Spearman's correlation coefficient is appropriate for ordinal data or for continuous data that doesn't meet the linear proportion requirement for Pearson's correlation coefficient.

Both inputs must be of the same length (no less than two), and need not to be constant, otherwise `StatisticsError` is raised.

Example with Kepler's laws of planetary motion :

```
>>> # Mercury, Venus, Earth, Mars, Jupiter, Saturn, Uranus, and  Neptune
>>> orbital_period = [88, 225, 365, 687, 4331, 10_756, 30_687, 60_190]    #␣
↪days
>>> dist_from_sun = [58, 108, 150, 228, 778, 1_400, 2_900, 4_500] # million km

>>> # Show that a perfect monotonic relationship exists
>>> correlation(orbital_period, dist_from_sun, method='ranked')
1.0

>>> # Observe that a linear relationship is imperfect
>>> round(correlation(orbital_period, dist_from_sun), 4)
0.9882

>>> # Demonstrate Kepler's third law: There is a linear correlation
>>> # between the square of the orbital period and the cube of the
>>> # distance from the sun.
>>> period_squared = [p * p for p in orbital_period]
>>> dist_cubed = [d * d * d for d in dist_from_sun]
>>> round(correlation(period_squared, dist_cubed), 4)
1.0
```

Ajouté dans la version 3.10.

Modifié dans la version 3.12 : Added support for Spearman's rank correlation coefficient.

`statistics.linear_regression`(*x*, *y*, /, *, *proportional=False*)

Return the slope and intercept of simple linear regression parameters estimated using ordinary least squares. Simple linear regression describes the relationship between an independent variable *x* and a dependent variable *y* in terms of this linear function :

*y = slope * x + intercept + noise*

where `slope` and `intercept` are the regression parameters that are estimated, and `noise` represents the variability of the data that was not explained by the linear regression (it is equal to the difference between predicted and actual values of the dependent variable).

Both inputs must be of the same length (no less than two), and the independent variable *x* cannot be constant ; otherwise a `StatisticsError` is raised.

For example, we can use the release dates of the Monty Python films to predict the cumulative number of Monty Python films that would have been produced by 2019 assuming that they had kept the pace.

```
>>> year = [1971, 1975, 1979, 1982, 1983]
>>> films_total = [1, 2, 3, 4, 5]
>>> slope, intercept = linear_regression(year, films_total)
>>> round(slope * 2019 + intercept)
16
```

If *proportional* is true, the independent variable *x* and the dependent variable *y* are assumed to be directly proportional. The data is fit to a line passing through the origin. Since the *intercept* will always be 0.0, the underlying linear function simplifies to :

*y = slope * x + noise*

Continuing the example from `correlation()`, we look to see how well a model based on major planets can predict the orbital distances for dwarf planets :

```
>>> model = linear_regression(period_squared, dist_cubed, proportional=True)
>>> slope = model.slope

>>> # Dwarf planets:   Pluto,  Eris,    Makemake, Haumea, Ceres
>>> orbital_periods = [90_560, 204_199, 111_845, 103_410, 1_680]  # days
>>> predicted_dist = [math.cbrt(slope * (p * p)) for p in orbital_periods]
>>> list(map(round, predicted_dist))
[5912, 10166, 6806, 6459, 414]

>>> [5_906, 10_152, 6_796, 6_450, 414]  # actual distance in million km
[5906, 10152, 6796, 6450, 414]
```

Ajouté dans la version 3.10.

Modifié dans la version 3.11 : Added support for *proportional*.

9.7.5 Exceptions

Une seule exception est définie :

exception statistics.**StatisticsError**

Sous-classe de `ValueError` pour les exceptions liées aux statistiques.

9.7.6 Objets `NormalDist`

`NormalDist` est un outil permettant de créer et manipuler des lois normales de variables aléatoires. Cette classe gère la moyenne et l'écart-type d'un ensemble d'observations comme une seule entité.

Les lois normales apparaissent dans de très nombreuses applications des statistiques. Leur ubiquité découle du théorème central limite.

class statistics.**NormalDist**(*mu=0.0*, *sigma=1.0*)

Renvoie un nouvel objet *NormalDist* où *mu* représente la moyenne arithmétique et *sigma* l'écart-type.

Lève une erreur `StatisticsError` si *sigma* est négatif.

mean

Attribut en lecture seule correspondant à la moyenne arithmétique d'une loi normale.

median

Attribut en lecture seule correspondant à la médiane d'une loi normale.

mode

Attribut en lecture seule correspondant au mode d'une loi normale.

stdev

Attribut en lecture seule correspondant à l'écart-type d'une loi normale.

variance

Attribut en lecture seule correspondant à la variance d'une loi normale. La variance est égale au carré de l'écart-type.

classmethod from_samples(*data*)

Crée une instance de loi normale de paramètres *mu* et *sigma* estimés à partir de *data* en utilisant `fmean()` et `stdev()`.

data peut être n'importe quel *iterable* de valeurs pouvant être converties en `float`. Lève une erreur `StatisticsError` si *data* ne contient pas au moins deux éléments car il faut au moins un point pour estimer la moyenne et deux points pour estimer la variance.

samples(*n*, *, *seed=None*)

Génère *n* valeurs aléatoires suivant une loi normale de moyenne et écart-type connus. Renvoie une `list` de `float`.

Si *seed* est spécifié, sa valeur est utilisée pour initialiser une nouvelle instance du générateur de nombres aléatoires. Cela permet de produire des résultats reproductibles même dans un contexte de parallélisme par fils d'exécution.

Modifié dans la version 3.13.

Switched to a faster algorithm. To reproduce samples from previous versions, use `random.seed()` and `random.gauss()`.

pdf(*x*)

Calcule la vraisemblance qu'une variable aléatoire *X* soit proche de la valeur *x* à partir de la fonction de densité. Mathématiquement, cela correspond à la limite de la fraction `P(x <= X < x + dx) / dx` lorsque `dx` tend vers zéro.

The relative likelihood is computed as the probability of a sample occurring in a narrow range divided by the width of the range (hence the word "density"). Since the likelihood is relative to other points, its value can be greater than `1.0`.

cdf(*x*)

Calcule la probabilité qu'une variable aléatoire *X* soit inférieure ou égale à *x* à partir de la fonction de répartition. Mathématiquement, cela correspond à `P(X <= x)`.

inv_cdf(*p*)

Compute the inverse cumulative distribution function, also known as the quantile function or the percent-point function. Mathematically, it is written `x : P(X <= x) = p`.

Détermine la valeur *x* de la variable aléatoire *X* telle que la probabilité que la variable soit inférieure ou égale à cette valeur *x* est égale à *p*.

overlap(*other*)

Mesure le recouvrement entre deux lois normales. Renvoie une valeur réelle entre 0 et 1 indiquant l'aire du recouvrement de deux densités de probabilité.

quantiles(*n=4*)

Divise la loi normale entre *n* intervalles réels équiprobables. Renvoie une liste de `(n - 1)` quantiles séparant les intervalles.

Utilisez `n = 4` pour obtenir les quartiles (le défaut), `n = 10` pour obtenir les déciles et `n = 100` pour obtenir les centiles (ce qui produit 99 valeurs qui séparent *data* en 100 groupes de même taille).

zscore(*x*)

Compute the Standard Score describing *x* in terms of the number of standard deviations above or below the mean of the normal distribution : `(x - mean) / stdev`.

Ajouté dans la version 3.9.

Les instances de la classe `NormalDist` gèrent l'addition, la soustraction, la multiplication et la division par une constante. Ces opérations peuvent être utilisées pour la translation ou la mise à l'échelle, par exemple :

```
>>> temperature_february = NormalDist(5, 2.5)             # Celsius
>>> temperature_february * (9/5) + 32                     # Fahrenheit
NormalDist(mu=41.0, sigma=4.5)
```

Diviser une constante par une instance de `NormalDist` n'est pas pris en charge car le résultat ne serait pas une loi normale.

Étant donné que les lois normales sont issues des propriétés additives de variables indépendantes, il est possible d'ajouter ou de soustraite deux variables normales indépendantes représentées par des instances de `NormalDist`. Par exemple :

```
>>> birth_weights = NormalDist.from_samples([2.5, 3.1, 2.1, 2.4, 2.7, 3.5])
>>> drug_effects = NormalDist(0.4, 0.15)
>>> combined = birth_weights + drug_effects
>>> round(combined.mean, 1)
3.1
>>> round(combined.stdev, 1)
0.5
```

Ajouté dans la version 3.8.

9.7.7 Examples and Recipes

Classic probability problems

`NormalDist` permet de résoudre aisément des problèmes probabilistes classiques.

Par exemple, sachant que les scores aux examens SAT suivent une loi normale de moyenne 1060 et d'écart-type 195, déterminer le pourcentage d'étudiants dont les scores se situent entre 1100 et 1200, arrondi à l'entier le plus proche :

```
>>> sat = NormalDist(1060, 195)
>>> fraction = sat.cdf(1200 + 0.5) - sat.cdf(1100 - 0.5)
>>> round(fraction * 100.0, 1)
18.4
```

Déterminer les quartiles et les déciles des scores SAT :

```
>>> list(map(round, sat.quantiles()))
[928, 1060, 1192]
>>> list(map(round, sat.quantiles(n=10)))
[810, 896, 958, 1011, 1060, 1109, 1162, 1224, 1310]
```

Monte Carlo inputs for simulations

To estimate the distribution for a model that isn't easy to solve analytically, `NormalDist` can generate input samples for a Monte Carlo simulation :

```
>>> def model(x, y, z):
...     return (3*x + 7*x*y - 5*y) / (11 * z)
...
>>> n = 100_000
>>> X = NormalDist(10, 2.5).samples(n, seed=3652260728)
>>> Y = NormalDist(15, 1.75).samples(n, seed=4582495471)
>>> Z = NormalDist(50, 1.25).samples(n, seed=6582483453)
>>> quantiles(map(model, X, Y, Z))
[1.4591308524824727, 1.8035946855390597, 2.175091447274739]
```

Approximating binomial distributions

Normal distributions can be used to approximate Binomial distributions when the sample size is large and when the probability of a successful trial is near 50%.

Par exemple, 750 personnes assistent à une conférence sur le logiciel libre. Il y a deux salles pouvant chacune accueillir 500 personnes. Dans la première salle a lieu une présentation sur Python, dans l'autre une présentation à propos de Ruby. Lors des conférences passées, 65% des personnes ont préféré écouter les présentations sur Python. En supposant que les préférences de la population n'ont pas changé, quelle est la probabilité que la salle Python reste en-dessous de sa capacité d'accueil ?

```
>>> n = 750             # Sample size
>>> p = 0.65            # Preference for Python
>>> q = 1.0 - p         # Preference for Ruby
>>> k = 500             # Room capacity

>>> # Approximation using the cumulative normal distribution
>>> from math import sqrt
>>> round(NormalDist(mu=n*p, sigma=sqrt(n*p*q)).cdf(k + 0.5), 4)
0.8402

>>> # Exact solution using the cumulative binomial distribution
>>> from math import comb, fsum
>>> round(fsum(comb(n, r) * p**r * q**(n-r) for r in range(k+1)), 4)
0.8402

>>> # Approximation using a simulation
>>> from random import seed, binomialvariate
>>> seed(8675309)
>>> mean(binomialvariate(n, p) <= k for i in range(10_000))
0.8406
```

Naive bayesian classifier

Les lois normales interviennent souvent en apprentissage automatique.

Wikipedia has a nice example of a Naive Bayesian Classifier. The challenge is to predict a person's gender from measurements of normally distributed features including height, weight, and foot size.

Nous avons à notre disposition un jeu de données d'apprentissage contenant les mesures de huit personnes. On suppose que ces mesures suivent une loi normale. Nous pouvons donc synthétiser les données à l'aide de `NormalDist` :

```
>>> height_male = NormalDist.from_samples([6, 5.92, 5.58, 5.92])
>>> height_female = NormalDist.from_samples([5, 5.5, 5.42, 5.75])
>>> weight_male = NormalDist.from_samples([180, 190, 170, 165])
>>> weight_female = NormalDist.from_samples([100, 150, 130, 150])
>>> foot_size_male = NormalDist.from_samples([12, 11, 12, 10])
>>> foot_size_female = NormalDist.from_samples([6, 8, 7, 9])
```

Ensuite, nous rencontrons un nouvel individu dont nous connaissons les proportions mais pas le sexe :

```
>>> ht = 6.0        # height
>>> wt = 130        # weight
>>> fs = 8          # foot size
```

En partant d'une probabilité a priori de 50% d'être un homme ou une femme, nous calculons la probabilité a posteriori comme le produit de la probabilité antérieure et de la vraisemblance des différentes mesures étant donné le sexe :

```
>>> prior_male = 0.5
>>> prior_female = 0.5
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> posterior_male = (prior_male * height_male.pdf(ht) *
...                   weight_male.pdf(wt) * foot_size_male.pdf(fs))

>>> posterior_female = (prior_female * height_female.pdf(ht) *
...                     weight_female.pdf(wt) * foot_size_female.pdf(fs))
```

La prédiction finale est celle qui a la plus grande probabilité a posteriori. Cette approche est appelée maximum a posteriori ou MAP :

```
>>> 'male' if posterior_male > posterior_female else 'female'
'female'
```

CHAPITRE 10

Modules de programmation fonctionnelle

Les modules décrits dans ce chapitre fournissent des fonctions et des classes permettant d'adopter un style fonctionnel, ainsi que des manipulations sur des appelables.

Les modules suivants sont documentés dans ce chapitre :

10.1 `itertools` --- Functions creating iterators for efficient looping

Ce module implémente de nombreuses briques *d'itérateurs* inspirées par des éléments de APL, Haskell et SML. Toutes ont été retravaillées dans un format adapté à Python.

Ce module standardise un ensemble de base d'outils rapides et efficaces en mémoire qui peuvent être utilisés individuellement ou en les combinant. Ensemble, ils forment une « algèbre d'itérateurs » rendant possible la construction rapide et efficace d'outils spécialisés en Python.

Par exemple, SML fournit un outil de tabulation `tabulate(f)` qui produit une séquence `f(0), f(1), ...`. Le même résultat peut être obtenu en Python en combinant *`map()`* et *`count()`* pour former `map(f, count())`.

Itérateurs infinis :

Itérateur	Arguments	Résultats	Exemple
`count()`	`[start[, step]]`	`start, start+step, start+2*step, ...`	`count(10) → 10 11 12 13 14 ...`
`cycle()`	p	p0, p1, ... plast, p0, p1, ...	`cycle('ABCD') → A B C D A B C D ...`
`repeat()`	elem [,n]	*elem*, *elem*, *elem*, ... à l'infini ou jusqu'à n fois	`repeat(10, 3) → 10 10 10`

Itérateurs se terminant par la séquence d'entrée la plus courte :

Itérateur	Arguments	Résultats	Exemple
`accumulate()`	p [,func]	p0, p0+p1, p0+p1+p2, ...	`accumulate([1,2,3,4,5]) → 1 3 6 10 15`
`batched()`	p, n	(p0, p1, ..., p_n-1), ...	`batched('ABCDEFG', n=3) → ABC DEF G`
`chain()`	p, q, ...	p0, p1, ... plast, q0, q1, ...	`chain('ABC', 'DEF') → A B C D E F`
`chain.from_iterable()`	itérable	p0, p1, ... plast, q0, q1, ...	`chain.from_iterable(['ABC', 'DEF']) → A B C D E F`
`compress()`	data, selectors	(d[0] if s[0]), (d[1] if s[1]), ...	`compress('ABCDEF', [1,0,1,0,1,1]) → A C E F`
`dropwhile()`	`predicate, seq`	seq[n], seq[n+1], starting when predicate fails	`dropwhile(lambda x: x<5, [1,4,6,3,8]) → 6 3 8`
`filterfalse()`	`predicate, seq`	elements of seq where predicate(elem) fails	`filterfalse(lambda x: x<5, [1,4,6,3,8]) → 6 8`
`groupby()`	iterable[, key]	sous-itérateurs groupés par la valeur de *key(v)*	`groupby(['A','B','DEF'], len) → (1, A B) (3, DEF)`
`islice()`	seq, [start,] stop [, step]	éléments de `seq[start:stop:step]`	`islice('ABCDEFG', 2, None) → C D E F G`
`pairwise()`	itérable	(p[0], p[1]), (p[1], p[2])	`pairwise('ABCDEFG') → AB BC CD DE EF FG`
`starmap()`	func, seq	func(*seq[0]), func(*seq[1]), ...	`starmap(pow, [(2,5), (3,2), (10,3)]) → 32 9 1000`
`takewhile()`	`predicate, seq`	seq[0], seq[1], until predicate fails	`takewhile(lambda x: x<5, [1,4,6,3,8]) → 1 4`
`tee()`	it, n	*it1*, *it2*, ... *itn* sépare un itérateur en *n*	`tee('ABC', 2) → A B C, A B C`
`zip_longest()`	p, q, ...	(p[0], q[0]), (p[1], q[1]), ...	`zip_longest('ABCD', 'xy', fillvalue='-') → Ax By C- D-`

Itérateurs combinatoires :

Itérateur	Arguments	Résultats
`product()`	p, q, ... [repeat=1]	produit cartésien, équivalent à une boucle *for* imbriquée
`permutations()`	p[, r]	*n*-uplets de longueur r, tous les ré-arrangements possibles, sans répétition d'éléments
`combinations()`	p, r	*n*-uplets de longueur r, ordonnés, sans répétition d'éléments
`combinations_with_replac`	p, r	*n*-uplets de longueur r, ordonnés, avec répétition d'éléments

Exemples	Résultats
`product('ABCD', repeat=2)`	`AA AB AC AD BA BB BC BD CA CB CC CD DA DB DC DD`
`permutations('ABCD', 2)`	`AB AC AD BA BC BD CA CB CD DA DB DC`
`combinations('ABCD', 2)`	`AB AC AD BC BD CD`
`combinations_with_replacement('ABCD', 2)`	`AA AB AC AD BB BC BD CC CD DD`

10.1.1 Itertool Functions

The following functions all construct and return iterators. Some provide streams of infinite length, so they should only be accessed by functions or loops that truncate the stream.

`itertools.accumulate(iterable[, function, *, initial=None])`

Make an iterator that returns accumulated sums or accumulated results from other binary functions.

The *function* defaults to addition. The *function* should accept two arguments, an accumulated total and a value from the *iterable*.

If an *initial* value is provided, the accumulation will start with that value and the output will have one more element than the input iterable.

À peu près équivalent à :

```
def accumulate(iterable, function=operator.add, *, initial=None):
    'Return running totals'
    # accumulate([1,2,3,4,5]) → 1 3 6 10 15
    # accumulate([1,2,3,4,5], initial=100) → 100 101 103 106 110 115
    # accumulate([1,2,3,4,5], operator.mul) → 1 2 6 24 120

    iterator = iter(iterable)
    total = initial
    if initial is None:
        try:
            total = next(iterator)
        except StopIteration:
            return

    yield total
    for element in iterator:
        total = function(total, element)
        yield total
```

To compute a running minimum, set *function* to `min()`. For a running maximum, set *function* to `max()`. Or for a running product, set *function* to `operator.mul()`. To build an amortization table, accumulate the interest and apply payments :

```
>>> data = [3, 4, 6, 2, 1, 9, 0, 7, 5, 8]
>>> list(accumulate(data, max))              # running maximum
[3, 4, 6, 6, 6, 9, 9, 9, 9, 9]
>>> list(accumulate(data, operator.mul))     # running product
[3, 12, 72, 144, 144, 1296, 0, 0, 0, 0]

# Amortize a 5% loan of 1000 with 10 annual payments of 90
>>> update = lambda balance, payment: round(balance * 1.05) - payment
>>> list(accumulate(repeat(90, 10), update, initial=1_000))
[1000, 960, 918, 874, 828, 779, 728, 674, 618, 559, 497]
```

Voir `functools.reduce()` pour une fonction similaire qui ne renvoie que la valeur accumulée finale.

Ajouté dans la version 3.2.

Modifié dans la version 3.3 : Added the optional *function* parameter.

Modifié dans la version 3.8 : Ajout du paramètre optionnel *initial*.

`itertools.batched(iterable, n, *, strict=False)`

Batch data from the *iterable* into tuples of length *n*. The last batch may be shorter than *n*.

If *strict* is true, will raise a `ValueError` if the final batch is shorter than *n*.

Loops over the input iterable and accumulates data into tuples up to size *n*. The input is consumed lazily, just

enough to fill a batch. The result is yielded as soon as the batch is full or when the input iterable is exhausted :

```
>>> flattened_data = ['roses', 'red', 'violets', 'blue', 'sugar', 'sweet']
>>> unflattened = list(batched(flattened_data, 2))
>>> unflattened
[('roses', 'red'), ('violets', 'blue'), ('sugar', 'sweet')]
```

À peu près équivalent à :

```
def batched(iterable, n, *, strict=False):
    # batched('ABCDEFG', 3) → ABC DEF G
    if n < 1:
        raise ValueError('n must be at least one')
    iterator = iter(iterable)
    while batch := tuple(islice(iterator, n)):
        if strict and len(batch) != n:
            raise ValueError('batched(): incomplete batch')
        yield batch
```

Ajouté dans la version 3.12.

Modifié dans la version 3.13 : Added the *strict* option.

itertools.**chain**(**iterables*)

Make an iterator that returns elements from the first iterable until it is exhausted, then proceeds to the next iterable, until all of the iterables are exhausted. This combines multiple data sources into a single iterator. Roughly equivalent to :

```
def chain(*iterables):
    # chain('ABC', 'DEF') → A B C D E F
    for iterable in iterables:
        yield from iterable
```

classmethod chain.**from_iterable**(*iterable*)

Constructeur alternatif pour `chain()`. Récupère des entrées chaînées depuis un unique itérable passé en argument, qui est évalué de manière paresseuse. À peu près équivalent à :

```
def from_iterable(iterables):
    # chain.from_iterable(['ABC', 'DEF']) → A B C D E F
    for iterable in iterables:
        yield from iterable
```

itertools.**combinations**(*iterable*, *r*)

Renvoie les combinaisons de longueur *r* de *iterable*.

The output is a subsequence of `product()` keeping only entries that are subsequences of the *iterable*. The length of the output is given by `math.comb()` which computes `n! / r! / (n - r)!` when `0 ≤ r ≤ n` or zero when `r > n`.

The combination tuples are emitted in lexicographic order according to the order of the input *iterable*. If the input *iterable* is sorted, the output tuples will be produced in sorted order.

Elements are treated as unique based on their position, not on their value. If the input elements are unique, there will be no repeated values within each combination.

À peu près équivalent à :

```
def combinations(iterable, r):
    # combinations('ABCD', 2) → AB AC AD BC BD CD
    # combinations(range(4), 3) → 012 013 023 123
```

(suite sur la page suivante)

(suite de la page précédente)

```
    pool = tuple(iterable)
    n = len(pool)
    if r > n:
        return
    indices = list(range(r))

    yield tuple(pool[i] for i in indices)
    while True:
        for i in reversed(range(r)):
            if indices[i] != i + n - r:
                break
        else:
            return
        indices[i] += 1
        for j in range(i+1, r):
            indices[j] = indices[j-1] + 1
        yield tuple(pool[i] for i in indices)
```

itertools.**combinations_with_replacement**(*iterable*, *r*)

Renvoyer les sous-séquences de longueur *r* des éléments de l'itérable *iterable* d'entrée, permettant aux éléments individuels d'être répétés plus d'une fois.

The output is a subsequence of `product()` that keeps only entries that are subsequences (with possible repeated elements) of the *iterable*. The number of subsequence returned is `(n + r - 1)! / r! / (n - 1)!` when `n > 0`.

The combination tuples are emitted in lexicographic order according to the order of the input *iterable*. if the input *iterable* is sorted, the output tuples will be produced in sorted order.

Elements are treated as unique based on their position, not on their value. If the input elements are unique, the generated combinations will also be unique.

À peu près équivalent à :

```
def combinations_with_replacement(iterable, r):
    # combinations_with_replacement('ABC', 2) → AA AB AC BB BC CC

    pool = tuple(iterable)
    n = len(pool)
    if not n and r:
        return
    indices = [0] * r

    yield tuple(pool[i] for i in indices)
    while True:
        for i in reversed(range(r)):
            if indices[i] != n - 1:
                break
        else:
            return
        indices[i:] = [indices[i] + 1] * (r - i)
        yield tuple(pool[i] for i in indices)
```

Ajouté dans la version 3.1.

itertools.**compress**(*data*, *selectors*)

Make an iterator that returns elements from *data* where the corresponding element in *selectors* is true. Stops when either the *data* or *selectors* iterables have been exhausted. Roughly equivalent to :

```
def compress(data, selectors):
    # compress('ABCDEF', [1,0,1,0,1,1]) → A C E F
    return (datum for datum, selector in zip(data, selectors) if selector)
```

Ajouté dans la version 3.1.

itertools.**count**(*start=0*, *step=1*)

Make an iterator that returns evenly spaced values beginning with *start*. Can be used with `map()` to generate consecutive data points or with `zip()` to add sequence numbers. Roughly equivalent to :

```
def count(start=0, step=1):
    # count(10) → 10 11 12 13 14 ...
    # count(2.5, 0.5) → 2.5 3.0 3.5 ...
    n = start
    while True:
        yield n
        n += step
```

When counting with floating-point numbers, better accuracy can sometimes be achieved by substituting multiplicative code such as : `(start + step * i for i in count())`.

Modifié dans la version 3.1 : Ajout de l'argument *step* et ajout du support pour les arguments non-entiers.

itertools.**cycle**(*iterable*)

Make an iterator returning elements from the *iterable* and saving a copy of each. When the iterable is exhausted, return elements from the saved copy. Repeats indefinitely. Roughly equivalent to :

```
def cycle(iterable):
    # cycle('ABCD') → A B C D A B C D A B C D ...

    saved = []
    for element in iterable:
        yield element
        saved.append(element)

    while saved:
        for element in saved:
            yield element
```

This itertool may require significant auxiliary storage (depending on the length of the iterable).

itertools.**dropwhile**(*predicate*, *iterable*)

Make an iterator that drops elements from the *iterable* while the *predicate* is true and afterwards returns every element. Roughly equivalent to :

```
def dropwhile(predicate, iterable):
    # dropwhile(lambda x: x<5, [1,4,6,3,8]) → 6 3 8

    iterator = iter(iterable)
    for x in iterator:
        if not predicate(x):
            yield x
            break

    for x in iterator:
        yield x
```

Note this does not produce *any* output until the predicate first becomes false, so this itertool may have a lengthy start-up time.

itertools.**filterfalse**(*predicate*, *iterable*)

Make an iterator that filters elements from the *iterable* returning only those for which the *predicate* returns a false value. If *predicate* is `None`, returns the items that are false. Roughly equivalent to :

```
def filterfalse(predicate, iterable):
    # filterfalse(lambda x: x<5, [1,4,6,3,8]) → 6 8

    if predicate is None:
        predicate = bool

    for x in iterable:
        if not predicate(x):
            yield x
```

itertools.**groupby**(*iterable*, *key=None*)

Crée un itérateur qui renvoie les clés et les groupes de l'itérable *iterable*. La clé *key* est une fonction qui génère une clé pour chaque élément. Si *key* n'est pas spécifiée ou est `None`, elle vaut par défaut une fonction d'identité qui renvoie l'élément sans le modifier. Généralement, l'itérable a besoin d'avoir ses éléments déjà classés selon cette même fonction de clé.

L'opération de `groupby()` est similaire au filtre `uniq` dans Unix. Elle génère un nouveau groupe à chaque fois que la valeur de la fonction *key* change (ce pourquoi il est souvent nécessaire d'avoir trié les données selon la même fonction de clé). Ce comportement est différent de celui de GROUP BY de SQL qui agrège les éléments sans prendre compte de leur ordre d'entrée.

Le groupe renvoyé est lui-même un itérateur qui partage l'itérable sous-jacent avec `groupby()`. Puisque que la source est partagée, quand l'objet `groupby()` est avancé, le groupe précédent n'est plus visible. Ainsi, si cette donnée doit être utilisée plus tard, elle doit être stockée comme une liste :

```
groups = []
uniquekeys = []
data = sorted(data, key=keyfunc)
for k, g in groupby(data, keyfunc):
    groups.append(list(g))      # Store group iterator as a list
    uniquekeys.append(k)
```

`groupby()` est à peu près équivalente à :

```
def groupby(iterable, key=None):
    # [k for k, g in groupby('AAAABBBCCDAABBB')] → A B C D A B
    # [list(g) for k, g in groupby('AAAABBBCCD')] → AAAA BBB CC D

    keyfunc = (lambda x: x) if key is None else key
    iterator = iter(iterable)
    exhausted = False

    def _grouper(target_key):
        nonlocal curr_value, curr_key, exhausted
        yield curr_value
        for curr_value in iterator:
            curr_key = keyfunc(curr_value)
            if curr_key != target_key:
                return
            yield curr_value
        exhausted = True

    try:
        curr_value = next(iterator)
    except StopIteration:
```

(suite sur la page suivante)

(suite de la page précédente)

```
        return
    curr_key = keyfunc(curr_value)

    while not exhausted:
        target_key = curr_key
        curr_group = _grouper(target_key)
        yield curr_key, curr_group
        if curr_key == target_key:
            for _ in curr_group:
                pass
```

itertools.**islice**(*iterable*, *stop*)

itertools.**islice**(*iterable*, *start*, *stop*[, *step*])

Make an iterator that returns selected elements from the iterable. Works like sequence slicing but does not support negative values for *start*, *stop*, or *step*.

If *start* is zero or `None`, iteration starts at zero. Otherwise, elements from the iterable are skipped until *start* is reached.

If *stop* is `None`, iteration continues until the input is exhausted, if at all. Otherwise, it stops at the specified position.

If *step* is `None`, the step defaults to one. Elements are returned consecutively unless *step* is set higher than one which results in items being skipped.

À peu près équivalent à :

```
def islice(iterable, *args):
    # islice('ABCDEFG', 2) → A B
    # islice('ABCDEFG', 2, 4) → C D
    # islice('ABCDEFG', 2, None) → C D E F G
    # islice('ABCDEFG', 0, None, 2) → A C E G

    s = slice(*args)
    start = 0 if s.start is None else s.start
    stop = s.stop
    step = 1 if s.step is None else s.step
    if start < 0 or (stop is not None and stop < 0) or step <= 0:
        raise ValueError

    indices = count() if stop is None else range(max(start, stop))
    next_i = start
    for i, element in zip(indices, iterable):
        if i == next_i:
            yield element
            next_i += step
```

If the input is an iterator, then fully consuming the *islice* advances the input iterator by `max(start, stop)` steps regardless of the *step* value.

itertools.**pairwise**(*iterable*)

Renvoie des paires successives d'éléments consécutifs de *iterable*.

En toute logique, il y a une paire de moins que d'éléments dans l'itérable. Aucune paire n'est renvoyée si l'itérable a zéro ou une valeur.

À peu près équivalent à :

```
def pairwise(iterable):
```

(suite sur la page suivante)

(suite de la page précédente)

```
    # pairwise('ABCDEFG') → AB BC CD DE EF FG

    iterator = iter(iterable)
    a = next(iterator, None)

    for b in iterator:
        yield a, b
        a = b
```

Ajouté dans la version 3.10.

itertools.**permutations**(*iterable*, *r=None*)

Return successive *r* length permutations of elements from the *iterable*.

Si *r* n'est pas spécifié ou vaut `None`, alors *r* a pour valeur la longueur de *iterable* et toutes les permutations de longueur *r* possibles sont générées.

The output is a subsequence of `product()` where entries with repeated elements have been filtered out. The length of the output is given by `math.perm()` which computes `n! / (n - r)!` when `0 ≤ r ≤ n` or zero when `r > n`.

The permutation tuples are emitted in lexicographic order according to the order of the input *iterable*. If the input *iterable* is sorted, the output tuples will be produced in sorted order.

Elements are treated as unique based on their position, not on their value. If the input elements are unique, there will be no repeated values within a permutation.

À peu près équivalent à :

```
def permutations(iterable, r=None):
    # permutations('ABCD', 2) → AB AC AD BA BC BD CA CB CD DA DB DC
    # permutations(range(3)) → 012 021 102 120 201 210

    pool = tuple(iterable)
    n = len(pool)
    r = n if r is None else r
    if r > n:
        return

    indices = list(range(n))
    cycles = list(range(n, n-r, -1))
    yield tuple(pool[i] for i in indices[:r])

    while n:
        for i in reversed(range(r)):
            cycles[i] -= 1
            if cycles[i] == 0:
                indices[i:] = indices[i+1:] + indices[i:i+1]
                cycles[i] = n - i
            else:
                j = cycles[i]
                indices[i], indices[-j] = indices[-j], indices[i]
                yield tuple(pool[i] for i in indices[:r])
                break
        else:
            return
```

itertools.**product**(**iterables*, *repeat=1*)

Cartesian product of the input iterables.

À peu près équivalent à des boucles *for* imbriquées dans une expression de générateur. Par exemple `product(A, B)` renvoie la même chose que `((x, y) for x in A for y in B)`.

Les boucles imbriquées tournent comme un compteur kilométrique avec l'élément le plus à droite avançant à chaque itération. Ce motif défini un ordre lexicographique afin que, si les éléments des itérables en l'entrée sont ordonnés, les *n*-uplets produits le sont aussi.

Pour générer le produit d'un itérable avec lui-même, spécifiez le nombre de répétitions avec le paramètre nommé optionnel *repeat*. Par exemple, `product(A, repeat=4)` est équivalent à `product(A, A, A, A)`.

Cette fonction est à peu près équivalente au code suivant, à la différence près que la vraie implémentation ne crée pas de résultats intermédiaires en mémoire :

```
def product(*iterables, repeat=1):
    # product('ABCD', 'xy') → Ax Ay Bx By Cx Cy Dx Dy
    # product(range(2), repeat=3) → 000 001 010 011 100 101 110 111

    if repeat < 0:
        raise ValueError('repeat argument cannot be negative')
    pools = [tuple(pool) for pool in iterables] * repeat

    result = [[]]
    for pool in pools:
        result = [x+[y] for x in result for y in pool]

    for prod in result:
        yield tuple(prod)
```

product() commence par consommer totalement les itérables qui lui sont passés et les conserve en mémoire pour générer les produits. Par conséquent, cette fonction ne sert que sur des itérables finis.

`itertools.`**`repeat`**(*object*[, *times*])

Crée un itérateur qui renvoie *object* à l'infini. S'exécute indéfiniment sauf si l'argument *times* est spécifié.

À peu près équivalent à :

```
def repeat(object, times=None):
    # repeat(10, 3) → 10 10 10
    if times is None:
        while True:
            yield object
    else:
        for i in range(times):
            yield object
```

Une utilisation courante de *repeat* est de fournir un flux constant de valeurs à *map* ou *zip* :

```
>>> list(map(pow, range(10), repeat(2)))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

`itertools.`**`starmap`**(*function*, *iterable*)

Make an iterator that computes the *function* using arguments obtained from the *iterable*. Used instead of *map()* when argument parameters have already been "pre-zipped" into tuples.

The difference between *map()* and *starmap()* parallels the distinction between `function(a,b)` and `function(*c)`. Roughly equivalent to :

```
def starmap(function, iterable):
    # starmap(pow, [(2,5), (3,2), (10,3)]) → 32 9 1000
    for args in iterable:
        yield function(*args)
```

itertools.**takewhile**(*predicate*, *iterable*)

Make an iterator that returns elements from the *iterable* as long as the *predicate* is true. Roughly equivalent to :

```
def takewhile(predicate, iterable):
    # takewhile(lambda x: x<5, [1,4,6,3,8]) → 1 4
    for x in iterable:
        if not predicate(x):
            break
        yield x
```

Note, the element that first fails the predicate condition is consumed from the input iterator and there is no way to access it. This could be an issue if an application wants to further consume the input iterator after *takewhile* has been run to exhaustion. To work around this problem, consider using more-itertools before_and_after() instead.

itertools.**tee**(*iterable*, *n=2*)

Renvoie *n* itérateurs indépendants depuis un unique itérable.

À peu près équivalent à :

```
def tee(iterable, n=2):
    if n < 0:
        raise ValueError
    if n == 0:
        return ()
    iterator = _tee(iterable)
    result = [iterator]
    for _ in range(n - 1):
        result.append(_tee(iterator))
    return tuple(result)

class _tee:

    def __init__(self, iterable):
        it = iter(iterable)
        if isinstance(it, _tee):
            self.iterator = it.iterator
            self.link = it.link
        else:
            self.iterator = it
            self.link = [None, None]

    def __iter__(self):
        return self

    def __next__(self):
        link = self.link
        if link[1] is None:
            link[0] = next(self.iterator)
            link[1] = [None, None]
        value, self.link = link
        return value
```

When the input *iterable* is already a tee iterator object, all members of the return tuple are constructed as if they had been produced by the upstream `tee()` call. This "flattening step" allows nested `tee()` calls to share the same underlying data chain and to have a single update step rather than a chain of calls.

The flattening property makes tee iterators efficiently peekable :

```
def lookahead(tee_iterator):
     "Return the next value without moving the input forward"
     [forked_iterator] = tee(tee_iterator, 1)
     return next(forked_iterator)
```

```
>>> iterator = iter('abcdef')
>>> [iterator] = tee(iterator, 1)    # Make the input peekable
>>> next(iterator)                   # Move the iterator forward
'a'
>>> lookahead(iterator)              # Check next value
'b'
>>> next(iterator)                   # Continue moving forward
'b'
```

`tee` iterators are not threadsafe. A `RuntimeError` may be raised when simultaneously using iterators returned by the same `tee()` call, even if the original *iterable* is threadsafe.

Cet outil peut avoir besoin d'un stockage auxiliaire important (en fonction de la taille des données temporaires nécessaires). En général, si un itérateur utilise la majorité ou toute la donnée avant qu'un autre itérateur ne commence, il est plus rapide d'utiliser `list()` à la place de `tee()`.

`itertools.`**`zip_longest`**`(`**iterables, fillvalue=None*`)`

Make an iterator that aggregates elements from each of the *iterables*.

If the iterables are of uneven length, missing values are filled-in with *fillvalue*. If not specified, *fillvalue* defaults to `None`.

Iteration continues until the longest iterable is exhausted.

À peu près équivalent à :

```
def zip_longest(*iterables, fillvalue=None):
    # zip_longest('ABCD', 'xy', fillvalue='-') → Ax By C- D-

    iterators = list(map(iter, iterables))
    num_active = len(iterators)
    if not num_active:
        return

    while True:
        values = []
        for i, iterator in enumerate(iterators):
            try:
                value = next(iterator)
            except StopIteration:
                num_active -= 1
                if not num_active:
                    return
                iterators[i] = repeat(fillvalue)
                value = fillvalue
            values.append(value)
        yield tuple(values)
```

If one of the iterables is potentially infinite, then the `zip_longest()` function should be wrapped with something that limits the number of calls (for example `islice()` or `takewhile()`).

10.1.2 Recettes *itertools*

Cette section présente des recettes pour créer une vaste boîte à outils en se servant des *itertools* existants comme des briques.

The primary purpose of the itertools recipes is educational. The recipes show various ways of thinking about individual tools — for example, that `chain.from_iterable` is related to the concept of flattening. The recipes also give ideas about ways that the tools can be combined — for example, how `starmap()` and `repeat()` can work together. The recipes also show patterns for using itertools with the *`operator`* and *`collections`* modules as well as with the built-in itertools such as `map()`, `filter()`, `reversed()`, and `enumerate()`.

A secondary purpose of the recipes is to serve as an incubator. The `accumulate()`, `compress()`, and `pairwise()` itertools started out as recipes. Currently, the `sliding_window()`, `iter_index()`, and `sieve()` recipes are being tested to see whether they prove their worth.

Substantially all of these recipes and many, many others can be installed from the more-itertools project found on the Python Package Index :

```
python -m pip install more-itertools
```

Many of the recipes offer the same high performance as the underlying toolset. Superior memory performance is kept by processing elements one at a time rather than bringing the whole iterable into memory all at once. Code volume is kept small by linking the tools together in a functional style. High speed is retained by preferring "vectorized" building blocks over the use of for-loops and *generators* which incur interpreter overhead.

```
from collections import Counter, deque
from contextlib import suppress
from functools import reduce
from math import comb, prod, sumprod, isqrt
from operator import itemgetter, getitem, mul, neg

def take(n, iterable):
    "Return first n items of the iterable as a list."
    return list(islice(iterable, n))

def prepend(value, iterable):
    "Prepend a single value in front of an iterable."
    # prepend(1, [2, 3, 4]) → 1 2 3 4
    return chain([value], iterable)

def tabulate(function, start=0):
    "Return function(0), function(1), ..."
    return map(function, count(start))

def repeatfunc(function, times=None, *args):
    "Repeat calls to a function with specified arguments."
    if times is None:
        return starmap(function, repeat(args))
    return starmap(function, repeat(args, times))

def flatten(list_of_lists):
    "Flatten one level of nesting."
    return chain.from_iterable(list_of_lists)

def ncycles(iterable, n):
    "Returns the sequence elements n times."
    return chain.from_iterable(repeat(tuple(iterable), n))

def loops(n):
    "Loop n times. Like range(n) but without creating integers."
```

(suite sur la page suivante)

(suite de la page précédente)

```
    # for _ in loops(100): ...
    return repeat(None, n)

def tail(n, iterable):
    "Return an iterator over the last n items."
    # tail(3, 'ABCDEFG') → E F G
    return iter(deque(iterable, maxlen=n))

def consume(iterator, n=None):
    "Advance the iterator n-steps ahead. If n is None, consume entirely."
    # Use functions that consume iterators at C speed.
    if n is None:
        deque(iterator, maxlen=0)
    else:
        next(islice(iterator, n, n), None)

def nth(iterable, n, default=None):
    "Returns the nth item or a default value."
    return next(islice(iterable, n, None), default)

def quantify(iterable, predicate=bool):
    "Given a predicate that returns True or False, count the True results."
    return sum(map(predicate, iterable))

def first_true(iterable, default=False, predicate=None):
    "Returns the first true value or the *default* if there is no true value."
    # first_true([a,b,c], x) → a or b or c or x
    # first_true([a,b], x, f) → a if f(a) else b if f(b) else x
    return next(filter(predicate, iterable), default)

def all_equal(iterable, key=None):
    "Returns True if all the elements are equal to each other."
    # all_equal('4□□□□', key=int) → True
    return len(take(2, groupby(iterable, key))) <= 1

def unique_justseen(iterable, key=None):
    "Yield unique elements, preserving order. Remember only the element just seen."
    # unique_justseen('AAAABBBCCDAABBB') → A B C D A B
    # unique_justseen('ABBcCAD', str.casefold) → A B c A D
    if key is None:
        return map(itemgetter(0), groupby(iterable))
    return map(next, map(itemgetter(1), groupby(iterable, key)))

def unique_everseen(iterable, key=None):
    "Yield unique elements, preserving order. Remember all elements ever seen."
    # unique_everseen('AAAABBBCCDAABBB') → A B C D
    # unique_everseen('ABBcCAD', str.casefold) → A B c D
    seen = set()
    if key is None:
        for element in filterfalse(seen.__contains__, iterable):
            seen.add(element)
            yield element
    else:
        for element in iterable:
            k = key(element)
            if k not in seen:
```

(suite sur la page suivante)

(suite de la page précédente)

```
                seen.add(k)
                yield element

def unique(iterable, key=None, reverse=False):
   "Yield unique elements in sorted order. Supports unhashable inputs."
   # unique([[1, 2], [3, 4], [1, 2]]) → [1, 2] [3, 4]
   sequenced = sorted(iterable, key=key, reverse=reverse)
   return unique_justseen(sequenced, key=key)

def sliding_window(iterable, n):
    "Collect data into overlapping fixed-length chunks or blocks."
    # sliding_window('ABCDEFG', 4) → ABCD BCDE CDEF DEFG
    iterator = iter(iterable)
    window = deque(islice(iterator, n - 1), maxlen=n)
    for x in iterator:
        window.append(x)
        yield tuple(window)

def grouper(iterable, n, *, incomplete='fill', fillvalue=None):
    "Collect data into non-overlapping fixed-length chunks or blocks."
    # grouper('ABCDEFG', 3, fillvalue='x') → ABC DEF Gxx
    # grouper('ABCDEFG', 3, incomplete='strict') → ABC DEF ValueError
    # grouper('ABCDEFG', 3, incomplete='ignore') → ABC DEF
    iterators = [iter(iterable)] * n
    match incomplete:
        case 'fill':
            return zip_longest(*iterators, fillvalue=fillvalue)
        case 'strict':
            return zip(*iterators, strict=True)
        case 'ignore':
            return zip(*iterators)
        case _:
            raise ValueError('Expected fill, strict, or ignore')

def roundrobin(*iterables):
    "Visit input iterables in a cycle until each is exhausted."
    # roundrobin('ABC', 'D', 'EF') → A D E B F C
    # Algorithm credited to George Sakkis
    iterators = map(iter, iterables)
    for num_active in range(len(iterables), 0, -1):
        iterators = cycle(islice(iterators, num_active))
        yield from map(next, iterators)

def subslices(seq):
    "Return all contiguous non-empty subslices of a sequence."
    # subslices('ABCD') → A AB ABC ABCD B BC BCD C CD D
    slices = starmap(slice, combinations(range(len(seq) + 1), 2))
    return map(getitem, repeat(seq), slices)

def iter_index(iterable, value, start=0, stop=None):
    "Return indices where a value occurs in a sequence or iterable."
    # iter_index('AABCADEAF', 'A') → 0 1 4 7
    seq_index = getattr(iterable, 'index', None)
    if seq_index is None:
        iterator = islice(iterable, start, stop)
        for i, element in enumerate(iterator, start):
```

(suite sur la page suivante)

(suite de la page précédente)

```
            if element is value or element == value:
                yield i
    else:
        stop = len(iterable) if stop is None else stop
        i = start
        with suppress(ValueError):
            while True:
                yield (i := seq_index(value, i, stop))
                i += 1

def iter_except(function, exception, first=None):
    "Convert a call-until-exception interface to an iterator interface."
    # iter_except(d.popitem, KeyError) → non-blocking dictionary iterator
    with suppress(exception):
        if first is not None:
            yield first()
        while True:
            yield function()
```

The following recipes have a more mathematical flavor :

```
def powerset(iterable):
    "Subsequences of the iterable from shortest to longest."
    # powerset([1,2,3]) → () (1,) (2,) (3,) (1,2) (1,3) (2,3) (1,2,3)
    s = list(iterable)
    return chain.from_iterable(combinations(s, r) for r in range(len(s)+1))

def sum_of_squares(iterable):
    "Add up the squares of the input values."
    # sum_of_squares([10, 20, 30]) → 1400
    return sumprod(*tee(iterable))

def reshape(matrix, columns):
    "Reshape a 2-D matrix to have a given number of columns."
    # reshape([(0, 1), (2, 3), (4, 5)], 3) →  (0, 1, 2), (3, 4, 5)
    return batched(chain.from_iterable(matrix), columns, strict=True)

def transpose(matrix):
    "Swap the rows and columns of a 2-D matrix."
    # transpose([(1, 2, 3), (11, 22, 33)]) → (1, 11) (2, 22) (3, 33)
    return zip(*matrix, strict=True)

def matmul(m1, m2):
    "Multiply two matrices."
    # matmul([(7, 5), (3, 5)], [(2, 5), (7, 9)]) → (49, 80), (41, 60)
    n = len(m2[0])
    return batched(starmap(sumprod, product(m1, transpose(m2))), n)

def convolve(signal, kernel):
    """Discrete linear convolution of two iterables.
    Equivalent to polynomial multiplication.

    Convolutions are mathematically commutative; however, the inputs are
    evaluated differently.  The signal is consumed lazily and can be
    infinite. The kernel is fully consumed before the calculations begin.
```

(suite sur la page suivante)

(suite de la page précédente)

```
    Article:  https://betterexplained.com/articles/intuitive-convolution/
    Video:    https://www.youtube.com/watch?v=KuXjwB4LzSA
    """
    # convolve([1, -1, -20], [1, -3]) → 1 -4 -17 60
    # convolve(data, [0.25, 0.25, 0.25, 0.25]) → Moving average (blur)
    # convolve(data, [1/2, 0, -1/2]) → 1st derivative estimate
    # convolve(data, [1, -2, 1]) → 2nd derivative estimate
    kernel = tuple(kernel)[::-1]
    n = len(kernel)
    padded_signal = chain(repeat(0, n-1), signal, repeat(0, n-1))
    windowed_signal = sliding_window(padded_signal, n)
    return map(sumprod, repeat(kernel), windowed_signal)

def polynomial_from_roots(roots):
    """Compute a polynomial's coefficients from its roots.

       (x - 5) (x + 4) (x - 3)  expands to:   x³ -4x² -17x + 60
    """
    # polynomial_from_roots([5, -4, 3]) → [1, -4, -17, 60]
    factors = zip(repeat(1), map(neg, roots))
    return list(reduce(convolve, factors, [1]))

def polynomial_eval(coefficients, x):
    """Evaluate a polynomial at a specific value.

    Computes with better numeric stability than Horner's method.
    """
    # Evaluate x³ -4x² -17x + 60 at x = 5
    # polynomial_eval([1, -4, -17, 60], x=5) → 0
    n = len(coefficients)
    if not n:
        return type(x)(0)
    powers = map(pow, repeat(x), reversed(range(n)))
    return sumprod(coefficients, powers)

def polynomial_derivative(coefficients):
    """Compute the first derivative of a polynomial.

       f(x)  =  x³ -4x² -17x + 60
       f'(x) = 3x² -8x  -17
    """
    # polynomial_derivative([1, -4, -17, 60]) → [3, -8, -17]
    n = len(coefficients)
    powers = reversed(range(1, n))
    return list(map(mul, coefficients, powers))

def sieve(n):
    "Primes less than n."
    # sieve(30) → 2 3 5 7 11 13 17 19 23 29
    if n > 2:
        yield 2
    data = bytearray((0, 1)) * (n // 2)
    for p in iter_index(data, 1, start=3, stop=isqrt(n) + 1):
        data[p*p : n : p+p] = bytes(len(range(p*p, n, p+p)))
    yield from iter_index(data, 1, start=3)
```

(suite sur la page suivante)

(suite de la page précédente)

```
def factor(n):
    "Prime factors of n."
    # factor(99) → 3 3 11
    # factor(1_000_000_000_000_007) → 47 59 360620266859
    # factor(1_000_000_000_000_403) → 1000000000000403
    for prime in sieve(isqrt(n) + 1):
        while not n % prime:
            yield prime
            n //= prime
            if n == 1:
                return
    if n > 1:
        yield n

def is_prime(n):
    "Return True if n is prime."
    # is_prime(1_000_000_000_000_403) → True
    return n > 1 and next(factor(n)) == n

def totient(n):
    "Count of natural numbers up to n that are coprime to n."
    # https://mathworld.wolfram.com/TotientFunction.html
    # totient(12) → 4 because len([1, 5, 7, 11]) == 4
    for prime in set(factor(n)):
        n -= n // prime
    return n

def multinomial(*counts):
    "Number of distinct arrangements of a multiset."
    # Counter('abracadabra').values() -> 5 2 1 1 2
    # multinomial(5, 2, 1, 1, 2) → 83160
    return prod(map(comb, accumulate(counts), counts))
```

10.2 `functools` --- Higher-order functions and operations on callable objects

Code source : Lib/functools.py

Le module *functools* concerne les fonctions d'ordre supérieur : des fonctions qui agissent sur, ou renvoient, d'autres fonctions. En général, tout objet appelable peut être considéré comme une fonction dans la description de ce module.

Le module *functools* définit les fonctions suivantes :

`@functools.cache(user_function)`

Fonction de cache très simple et sans limite de taille. Cette technique est parfois appelée « mémoïsation ».

Returns the same as `lru_cache(maxsize=None)`, creating a thin wrapper around a dictionary lookup for the function arguments. Because it never needs to evict old values, this is smaller and faster than *lru_cache()* with a size limit.

Par exemple :

```
@cache
def factorial(n):
    return n * factorial(n-1) if n else 1
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> factorial(10)       # no previously cached result, makes 11 recursive calls
3628800
>>> factorial(5)        # just looks up cached value result
120
>>> factorial(12)       # makes two new recursive calls, the other 10 are cached
479001600
```

The cache is threadsafe so that the wrapped function can be used in multiple threads. This means that the underlying data structure will remain coherent during concurrent updates.

It is possible for the wrapped function to be called more than once if another thread makes an additional call before the initial call has been completed and cached.

Ajouté dans la version 3.9.

`@functools.cached_property`(*func*)

Transform a method of a class into a property whose value is computed once and then cached as a normal attribute for the life of the instance. Similar to `property()`, with the addition of caching. Useful for expensive computed properties of instances that are otherwise effectively immutable.

Exemple :

```
class DataSet:

    def __init__(self, sequence_of_numbers):
        self._data = tuple(sequence_of_numbers)

    @cached_property
    def stdev(self):
        return statistics.stdev(self._data)
```

The mechanics of `cached_property()` are somewhat different from `property()`. A regular property blocks attribute writes unless a setter is defined. In contrast, a *cached_property* allows writes.

The *cached_property* decorator only runs on lookups and only when an attribute of the same name doesn't exist. When it does run, the *cached_property* writes to the attribute with the same name. Subsequent attribute reads and writes take precedence over the *cached_property* method and it works like a normal attribute.

The cached value can be cleared by deleting the attribute. This allows the *cached_property* method to run again.

The *cached_property* does not prevent a possible race condition in multi-threaded usage. The getter function could run more than once on the same instance, with the latest run setting the cached value. If the cached property is idempotent or otherwise not harmful to run more than once on an instance, this is fine. If synchronization is needed, implement the necessary locking inside the decorated getter function or around the cached property access.

Note, this decorator interferes with the operation of **PEP 412** key-sharing dictionaries. This means that instance dictionaries can take more space than usual.

Also, this decorator requires that the `__dict__` attribute on each instance be a mutable mapping. This means it will not work with some types, such as metaclasses (since the `__dict__` attributes on type instances are read-only proxies for the class namespace), and those that specify `__slots__` without including `__dict__` as one of the defined slots (as such classes don't provide a `__dict__` attribute at all).

If a mutable mapping is not available or if space-efficient key sharing is desired, an effect similar to `cached_property()` can also be achieved by stacking `property()` on top of `lru_cache()`. See faq-cache-method-calls for more details on how this differs from `cached_property()`.

Ajouté dans la version 3.8.

Modifié dans la version 3.12 : Prior to Python 3.12, `cached_property` included an undocumented lock to ensure that in multi-threaded usage the getter function was guaranteed to run only once per instance. However,

the lock was per-property, not per-instance, which could result in unacceptably high lock contention. In Python 3.12+ this locking is removed.

functools.**cmp_to_key**(*func*)

Transforme une fonction de comparaison à l'ancienne en une *fonction clé*. Utilisé avec des outils qui acceptent des fonctions clef (comme `sorted()`, `min()`, `max()`, `heapq.nlargest()`, `heapq.nsmallest()`, `itertools.groupby()`). Cette fonction est destinée au portage de fonctions python 2 utilisant des fonctions de comparaison vers Python 3.

A comparison function is any callable that accepts two arguments, compares them, and returns a negative number for less-than, zero for equality, or a positive number for greater-than. A key function is a callable that accepts one argument and returns another value to be used as the sort key.

Exemple :

```
sorted(iterable, key=cmp_to_key(locale.strcoll))  # locale-aware sort order
```

Pour des exemples de tris et un bref tutoriel, consultez sortinghowto.

Ajouté dans la version 3.2.

@functools.**lru_cache**(*user_function*)

@functools.**lru_cache**(*maxsize=128*, *typed=False*)

Décorateur qui englobe une fonction avec un appelable mémoïsant qui enregistre jusqu'à *maxsize* appels récents. Cela peut gagner du temps quand une fonction coûteuse en ressources est souvent appelée avec les mêmes arguments.

The cache is threadsafe so that the wrapped function can be used in multiple threads. This means that the underlying data structure will remain coherent during concurrent updates.

It is possible for the wrapped function to be called more than once if another thread makes an additional call before the initial call has been completed and cached.

Since a dictionary is used to cache results, the positional and keyword arguments to the function must be *hashable*.

Distinct argument patterns may be considered to be distinct calls with separate cache entries. For example, `f(a=1, b=2)` and `f(b=2, a=1)` differ in their keyword argument order and may have two separate cache entries.

Si *user_function* est défini, ce doit être un appelable. Ceci permet à *lru_cache* d'être appliqué directement sur une fonction de l'utilisateur, sans préciser *maxsize* (qui est alors défini à sa valeur par défaut, 128) :

```
@lru_cache
def count_vowels(sentence):
    return sum(sentence.count(vowel) for vowel in 'AEIOUaeiou')
```

Si *maxsize* est à `None`, la fonctionnalité LRU est désactivée et le cache peut grossir sans limite.

If *typed* is set to true, function arguments of different types will be cached separately. If *typed* is false, the implementation will usually regard them as equivalent calls and only cache a single result. (Some types such as *str* and *int* may be cached separately even when *typed* is false.)

Note, type specificity applies only to the function's immediate arguments rather than their contents. The scalar arguments, `Decimal(42)` and `Fraction(42)` are be treated as distinct calls with distinct results. In contrast, the tuple arguments `('answer', Decimal(42))` and `('answer', Fraction(42))` are treated as equivalent.

The wrapped function is instrumented with a `cache_parameters()` function that returns a new `dict` showing the values for *maxsize* and *typed*. This is for information purposes only. Mutating the values has no effect.

To help measure the effectiveness of the cache and tune the *maxsize* parameter, the wrapped function is instrumented with a `cache_info()` function that returns a *named tuple* showing *hits*, *misses*, *maxsize* and *currsize*.

Le décorateur fournit également une fonction `cache_clear()` pour vider ou invalider le cache.

La fonction sous-jacente originale est accessible à travers l'attribut `__wrapped__`. Ceci est utile pour l'introspection, pour outrepasser le cache, ou pour ré-englober la fonction avec un cache différent.

The cache keeps references to the arguments and return values until they age out of the cache or until the cache is cleared.

If a method is cached, the `self` instance argument is included in the cache. See faq-cache-method-calls

An LRU (least recently used) cache works best when the most recent calls are the best predictors of upcoming calls (for example, the most popular articles on a news server tend to change each day). The cache's size limit assures that the cache does not grow without bound on long-running processes such as web servers.

In general, the LRU cache should only be used when you want to reuse previously computed values. Accordingly, it doesn't make sense to cache functions with side-effects, functions that need to create distinct mutable objects on each call (such as generators and async functions), or impure functions such as time() or random().

Exemple d'un cache LRU pour du contenu web statique :

```
@lru_cache(maxsize=32)
def get_pep(num):
    'Retrieve text of a Python Enhancement Proposal'
    resource = f'https://peps.python.org/pep-{num:04d}'
    try:
        with urllib.request.urlopen(resource) as s:
            return s.read()
    except urllib.error.HTTPError:
        return 'Not Found'

>>> for n in 8, 290, 308, 320, 8, 218, 320, 279, 289, 320, 9991:
...     pep = get_pep(n)
...     print(n, len(pep))

>>> get_pep.cache_info()
CacheInfo(hits=3, misses=8, maxsize=32, currsize=8)
```

Exemple de calcul efficace de la suite de Fibonacci en utilisant un cache pour implémenter la technique de programmation dynamique :

```
@lru_cache(maxsize=None)
def fib(n):
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)

>>> [fib(n) for n in range(16)]
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610]

>>> fib.cache_info()
CacheInfo(hits=28, misses=16, maxsize=None, currsize=16)
```

Ajouté dans la version 3.2.

Modifié dans la version 3.3 : L'option *typed* a été ajoutée.

Modifié dans la version 3.8 : Ajout de l'option *user_function*.

Modifié dans la version 3.9 : Added the function `cache_parameters()`

`@functools.total_ordering`

A partir d'une classe définissant une ou plusieurs méthodes de comparaison riches, ce décorateur de classe fournit le reste. Ceci simplifie l'effort à fournir dans la spécification de toutes les opérations de comparaison riche :

La classe doit définir au moins une de ces méthodes `__lt__()`, `__le__()`, `__gt__()`, ou `__ge__()`. De plus, la classe doit fournir une méthode `__eq__()`.

Par exemple :

```
@total_ordering
class Student:
    def _is_valid_operand(self, other):
        return (hasattr(other, "lastname") and
                hasattr(other, "firstname"))
    def __eq__(self, other):
        if not self._is_valid_operand(other):
            return NotImplemented
        return ((self.lastname.lower(), self.firstname.lower()) ==
                (other.lastname.lower(), other.firstname.lower()))
    def __lt__(self, other):
        if not self._is_valid_operand(other):
            return NotImplemented
        return ((self.lastname.lower(), self.firstname.lower()) <
                (other.lastname.lower(), other.firstname.lower()))
```

Note

Même si ce décorateur permet de créer des types ordonnables facilement, cela vient avec un *coût* d'exécution et des traces d'exécution complexes pour les méthodes de comparaison dérivées. Si des tests de performances le révèlent comme un goulot d'étranglement, l'implémentation manuelle des six méthodes de comparaison riches résoudra normalement vos problèmes de rapidité.

Note

This decorator makes no attempt to override methods that have been declared in the class *or its superclasses*. Meaning that if a superclass defines a comparison operator, *total_ordering* will not implement it again, even if the original method is abstract.

Ajouté dans la version 3.2.

Modifié dans la version 3.4 : Returning `NotImplemented` from the underlying comparison function for unrecognised types is now supported.

`functools.`**`Placeholder`**

A singleton object used as a sentinel to reserve a place for positional arguments when calling *`partial()`* and *`partialmethod()`*.

Ajouté dans la version 3.14.

`functools.`**`partial`**(*func*, /, **args*, ***keywords*)

Retourne un nouvel *objet partiel* qui, quand il est appelé, fonctionne comme *func* appelée avec les arguments positionnels *args* et les arguments nommés *keywords*. Si plus d'arguments sont fournis à l'appel, ils sont ajoutés à *args*. Si plus d'arguments nommés sont fournis, ils étendent et surchargent *keywords*. À peu près équivalent à :

```
def partial(func, /, *args, **keywords):
    def newfunc(*more_args, **more_keywords):
        return func(*args, *more_args, **(keywords | more_keywords))
    newfunc.func = func
    newfunc.args = args
    newfunc.keywords = keywords
```

(suite sur la page suivante)

(suite de la page précédente)

```
    return newfunc
```

The `partial()` function is used for partial function application which "freezes" some portion of a function's arguments and/or keywords resulting in a new object with a simplified signature. For example, *partial()* can be used to create a callable that behaves like the *int()* function where the *base* argument defaults to `2` :

```
>>> basetwo = partial(int, base=2)
>>> basetwo.__doc__ = 'Convert base 2 string to an int.'
>>> basetwo('10010')
18
```

If *Placeholder* sentinels are present in *args*, they will be filled first when `partial()` is called. This makes it possible to pre-fill any positional argument with a call to `partial()` ; without `Placeholder`, only the chosen number of leading positional arguments can be pre-filled.

If any `Placeholder` sentinels are present, all must be filled at call time :

```
>>> say_to_world = partial(print, Placeholder, Placeholder, "world!")
>>> say_to_world('Hello', 'dear')
Hello dear world!
```

Calling `say_to_world('Hello')` raises a *TypeError*, because only one positional argument is provided, but there are two placeholders that must be filled in.

If `partial()` is applied to an existing `partial()` object, `Placeholder` sentinels of the input object are filled in with new positional arguments. A placeholder can be retained by inserting a new `Placeholder` sentinel to the place held by a previous `Placeholder` :

```
>>> from functools import partial, Placeholder as _
>>> remove = partial(str.replace, _, _, '')
>>> message = 'Hello, dear dear world!'
>>> remove(message, ' dear')
'Hello, world!'
>>> remove_dear = partial(remove, _, ' dear')
>>> remove_dear(message)
'Hello, world!'
>>> remove_first_dear = partial(remove_dear, _, 1)
>>> remove_first_dear(message)
'Hello, dear world!'
```

`Placeholder` has no special treatment when used in a keyword argument to `partial()`.

Modifié dans la version 3.14 : Added support for *Placeholder* in positional arguments.

class functools.**partialmethod**(*func*, /, **args*, ***keywords*)

Retourne un nouveau descripteur *partialmethod* qui se comporte comme *partial* sauf qu'il est fait pour être utilisé comme une définition de méthode plutôt que d'être appelé directement.

func doit être un *descriptor* ou un appelable (les objets qui sont les deux, comme les fonction normales, sont gérés comme des descripteurs).

Quand *func* est un descripteur (comme une fonction Python normale, *classmethod()*, *staticmethod()*, `abstractmethod()` ou une autre instance de *partialmethod*), les appels à `__get__` sont délégués au descripteur sous-jacent, et un *objet partiel* approprié est renvoyé comme résultat.

Quand *func* est un appelable non-descripteur, une méthode liée appropriée est crée dynamiquement. Elle se comporte comme une fonction Python normale quand elle est utilisée comme méthode : l'argument *self* sera inséré comme premier argument positionnel, avant les *args* et *keywords* fournis au constructeur *partialmethod*.

Exemple :

```
>>> class Cell:
...     def __init__(self):
...         self._alive = False
...     @property
...     def alive(self):
...         return self._alive
...     def set_state(self, state):
...         self._alive = bool(state)
...     set_alive = partialmethod(set_state, True)
...     set_dead = partialmethod(set_state, False)
...
>>> c = Cell()
>>> c.alive
False
>>> c.set_alive()
>>> c.alive
True
```

Ajouté dans la version 3.4.

functools.**reduce**(*function*, *iterable*, /[, *initial*])

Apply *function* of two arguments cumulatively to the items of *iterable*, from left to right, so as to reduce the iterable to a single value. For example, `reduce(lambda x, y: x+y, [1, 2, 3, 4, 5])` calculates `((((1+2)+3)+4)+5)`. The left argument, *x*, is the accumulated value and the right argument, *y*, is the update value from the *iterable*. If the optional *initial* is present, it is placed before the items of the iterable in the calculation, and serves as a default when the iterable is empty. If *initial* is not given and *iterable* contains only one item, the first item is returned.

À peu près équivalent à :

```
initial_missing = object()

def reduce(function, iterable, /, initial=initial_missing):
    it = iter(iterable)
    if initial is initial_missing:
        value = next(it)
    else:
        value = initial
    for element in it:
        value = function(value, element)
    return value
```

Voir `itertools.accumulate()` pour un itérateur qui génère toutes les valeurs intermédiaires.

Modifié dans la version 3.14 : *initial* is now supported as a keyword argument.

@functools.**singledispatch**

Transforme une fonction en une *fonction générique single-dispatch*.

To define a generic function, decorate it with the `@singledispatch` decorator. When defining a function using `@singledispatch`, note that the dispatch happens on the type of the first argument :

```
>>> from functools import singledispatch
>>> @singledispatch
... def fun(arg, verbose=False):
...     if verbose:
...         print("Let me just say,", end=" ")
...     print(arg)
```

To add overloaded implementations to the function, use the `register()` attribute of the generic function,

which can be used as a decorator. For functions annotated with types, the decorator will infer the type of the first argument automatically :

```
>>> @fun.register
... def _(arg: int, verbose=False):
...     if verbose:
...         print("Strength in numbers, eh?", end=" ")
...     print(arg)
...
>>> @fun.register
... def _(arg: list, verbose=False):
...     if verbose:
...         print("Enumerate this:")
...     for i, elem in enumerate(arg):
...         print(i, elem)
```

types.UnionType and *typing.Union* can also be used :

```
>>> @fun.register
... def _(arg: int | float, verbose=False):
...     if verbose:
...         print("Strength in numbers, eh?", end=" ")
...     print(arg)
...
>>> from typing import Union
>>> @fun.register
... def _(arg: Union[list, set], verbose=False):
...     if verbose:
...         print("Enumerate this:")
...     for i, elem in enumerate(arg):
...         print(i, elem)
...
```

Pour le code qui n'utilise pas les indications de type, le type souhaité peut être passé explicitement en argument au décorateur :

```
>>> @fun.register(complex)
... def _(arg, verbose=False):
...     if verbose:
...         print("Better than complicated.", end=" ")
...     print(arg.real, arg.imag)
...
```

For code that dispatches on a collections type (e.g., `list`), but wants to typehint the items of the collection (e.g., `list[int]`), the dispatch type should be passed explicitly to the decorator itself with the typehint going into the function definition :

```
>>> @fun.register(list)
... def _(arg: list[int], verbose=False):
...     if verbose:
...         print("Enumerate this:")
...     for i, elem in enumerate(arg):
...         print(i, elem)
```

Note

At runtime the function will dispatch on an instance of a list regardless of the type contained within the list i.e. `[1,2,3]` will be dispatched the same as `["foo", "bar", "baz"]`. The annotation provided in

this example is for static type checkers only and has no runtime impact.

To enable registering *lambdas* and pre-existing functions, the `register()` attribute can also be used in a functional form :

```
>>> def nothing(arg, verbose=False):
...     print("Nothing.")
...
>>> fun.register(type(None), nothing)
```

The `register()` attribute returns the undecorated function. This enables decorator stacking, *pickling*, and the creation of unit tests for each variant independently :

```
>>> @fun.register(float)
... @fun.register(Decimal)
... def fun_num(arg, verbose=False):
...     if verbose:
...         print("Half of your number:", end=" ")
...     print(arg / 2)
...
>>> fun_num is fun
False
```

Quand elle est appelée, la fonction générique distribue sur le type du premier argument :

```
>>> fun("Hello, world.")
Hello, world.
>>> fun("test.", verbose=True)
Let me just say, test.
>>> fun(42, verbose=True)
Strength in numbers, eh? 42
>>> fun(['spam', 'spam', 'eggs', 'spam'], verbose=True)
Enumerate this:
0 spam
1 spam
2 eggs
3 spam
>>> fun(None)
Nothing.
>>> fun(1.23)
0.615
```

Where there is no registered implementation for a specific type, its method resolution order is used to find a more generic implementation. The original function decorated with `@singledispatch` is registered for the base `object` type, which means it is used if no better implementation is found.

If an implementation is registered to an *abstract base class*, virtual subclasses of the base class will be dispatched to that implementation :

```
>>> from collections.abc import Mapping
>>> @fun.register
... def _(arg: Mapping, verbose=False):
...     if verbose:
...         print("Keys & Values")
...     for key, value in arg.items():
...         print(key, "=>", value)
...
>>> fun({"a": "b"})
```

(suite sur la page suivante)

(suite de la page précédente)

```
a => b
```

To check which implementation the generic function will choose for a given type, use the `dispatch()` attribute :

```
>>> fun.dispatch(float)
<function fun_num at 0x1035a2840>
>>> fun.dispatch(dict)    # note: default implementation
<function fun at 0x103fe0000>
```

Pour accéder à toutes les implémentations enregistrées, utiliser l'attribut en lecture seule `registry` :

```
>>> fun.registry.keys()
dict_keys([<class 'NoneType'>, <class 'int'>, <class 'object'>,
          <class 'decimal.Decimal'>, <class 'list'>,
          <class 'float'>])
>>> fun.registry[float]
<function fun_num at 0x1035a2840>
>>> fun.registry[object]
<function fun at 0x103fe0000>
```

Ajouté dans la version 3.4.

Modifié dans la version 3.7 : The `register()` attribute now supports using type annotations.

Modifié dans la version 3.11 : The `register()` attribute now supports `types.UnionType` and `typing.Union` as type annotations.

class functools.**singledispatchmethod**(*func*)

Transforme une méthode en une *fonction générique single-dispatch*.

To define a generic method, decorate it with the `@singledispatchmethod` decorator. When defining a function using `@singledispatchmethod`, note that the dispatch happens on the type of the first non-*self* or non-*cls* argument :

```
class Negator:
    @singledispatchmethod
    def neg(self, arg):
        raise NotImplementedError("Cannot negate a")

    @neg.register
    def _(self, arg: int):
        return -arg

    @neg.register
    def _(self, arg: bool):
        return not arg
```

`@singledispatchmethod` supports nesting with other decorators such as `@classmethod`. Note that to allow for `dispatcher.register`, `singledispatchmethod` must be the *outer most* decorator. Here is the `Negator` class with the `neg` methods bound to the class, rather than an instance of the class :

```
class Negator:
    @singledispatchmethod
    @classmethod
    def neg(cls, arg):
        raise NotImplementedError("Cannot negate a")

    @neg.register
```

(suite sur la page suivante)

(suite de la page précédente)

```
    @classmethod
    def _(cls, arg: int):
        return -arg

    @neg.register
    @classmethod
    def _(cls, arg: bool):
        return not arg
```

The same pattern can be used for other similar decorators : `@staticmethod`, `@abstractmethod`, and others.

Ajouté dans la version 3.8.

`functools.update_wrapper`(*wrapper*, *wrapped*, *assigned=WRAPPER_ASSIGNMENTS*, *updated=WRAPPER_UPDATES*)

Update a *wrapper* function to look like the *wrapped* function. The optional arguments are tuples to specify which attributes of the original function are assigned directly to the matching attributes on the wrapper function and which attributes of the wrapper function are updated with the corresponding attributes from the original function. The default values for these arguments are the module level constants `WRAPPER_ASSIGNMENTS` (which assigns to the wrapper function's `__module__`, `__name__`, `__qualname__`, `__annotations__`, `__type_params__`, and `__doc__`, the documentation string) and `WRAPPER_UPDATES` (which updates the wrapper function's `__dict__`, i.e. the instance dictionary).

Pour autoriser l'accès à la fonction originale pour l'introspection ou à d'autres fins (par ex. outrepasser l'accès à un décorateur de cache comme `lru_cache()`), cette fonction ajoute automatiquement un attribut `__wrapped__` qui référence la fonction englobée.

La principale utilisation de cette fonction est dans les *décorateurs* qui renvoient une nouvelle fonction. Si la fonction crée n'est pas mise à jour, ses métadonnées refléteront sa définition dans le décorateur, au lieu de la définition originale, métadonnées souvent bien moins utiles.

`update_wrapper()` peut être utilisé avec des appelables autres que des fonctions. Tout attribut défini dans *assigned* ou *updated* qui ne sont pas l'objet englobé sont ignorés (cette fonction n'essaiera pas de les définir dans la fonction englobante). `AttributeError` est toujours levée si le fonction englobante elle même a des attributs non existants dans *updated*.

Modifié dans la version 3.2 : The `__wrapped__` attribute is now automatically added. The `__annotations__` attribute is now copied by default. Missing attributes no longer trigger an `AttributeError`.

Modifié dans la version 3.4 : L'attribut `__wrapped__` renvoie toujours la fonction englobée, même si cette fonction définit un attribut `__wrapped__`. (voir bpo-17482)

Modifié dans la version 3.12 : The `__type_params__` attribute is now copied by default.

`@functools.wraps`(*wrapped*, *assigned=WRAPPER_ASSIGNMENTS*, *updated=WRAPPER_UPDATES*)

Ceci est une fonction d'aide pour appeler `update_wrapper()` comme décorateur de fonction lors de la définition d'une fonction englobante. C'est équivalent à `partial(update_wrapper, wrapped=wrapped, assigned=assigned, updated=updated)`. Par exemple :

```
>>> from functools import wraps
>>> def my_decorator(f):
...     @wraps(f)
...     def wrapper(*args, **kwds):
...         print('Calling decorated function')
...         return f(*args, **kwds)
...     return wrapper
...
>>> @my_decorator
... def example():
...     """Docstring"""
```

(suite sur la page suivante)

(suite de la page précédente)

```
...     print('Called example function')
...
>>> example()
Calling decorated function
Called example function
>>> example.__name__
'example'
>>> example.__doc__
'Docstring'
```

Sans l'utilisation de cette usine à décorateur, le nom de la fonction d'exemple aurait été `'wrapper'`, et la chaîne de documentation de la fonction `example()` originale aurait été perdue.

10.2.1 Objets `partial`

Les objets *partial* sont des objets appelables créés par *partial()*. Ils ont trois attributs en lecture seule :

`partial.`**`func`**

Un objet ou une fonction appelable. Les appels à l'objet *partial* seront transmis à *func* avec les nouveaux arguments et mots-clés.

`partial.`**`args`**

Les arguments positionnels qui seront ajoutés avant les arguments fournis lors de l'appel d'un objet *partial*.

`partial.`**`keywords`**

Les arguments nommés qui seront fournis quand l'objet *partial* est appelé.

partial objects are like function objects in that they are callable, weak referenceable, and can have attributes. There are some important differences. For instance, the *__name__* and *__doc__* attributes are not created automatically.

10.3 `operator` — Opérateurs standards en tant que fonctions

Code source : Lib/operator.py

Le module *operator* fournit un ensemble de fonctions correspondant aux opérateurs natifs de Python. Par exemple, `operator.add(x, y)` correspond à l'expression `x+y`. Les noms de la plupart de ces fonctions sont ceux utilisés par les méthodes spéciales, sans les doubles tirets bas. Pour assurer la rétrocompatibilité, la plupart de ces noms ont une variante *avec* les doubles tirets bas ; la forme simple est cependant à privilégier pour des raisons de clarté.

Les fonctions sont divisées en différentes catégories selon l'opération effectuée : comparaison entre objets, opérations logiques, opérations mathématiques ou opérations sur séquences.

Les fonctions de comparaison s'appliquent à tous les objets, et leur nom vient des opérateurs de comparaison qu'elles implémentent :

`operator.`**`lt`**(*a*, *b*)

`operator.`**`le`**(*a*, *b*)

`operator.`**`eq`**(*a*, *b*)

`operator.`**`ne`**(*a*, *b*)

`operator.`**`ge`**(*a*, *b*)

`operator.`**`gt`**(*a*, *b*)

`operator.`**`__lt__`**(*a*, *b*)

`operator.`**`__le__`**(*a*, *b*)

`operator.`**`__eq__`**(*a*, *b*)

`operator.`**`__ne__`**(*a*, *b*)

`operator.`**`__ge__`**(*a*, *b*)

operator.**__gt__**(*a*, *b*)

Effectue une « comparaison riche » entre *a* et *b*. Plus précisément, `lt(a, b)` équivaut à `a < b`, `le(a, b)` équivaut à `a <= b`, `eq(a, b)` équivaut à `a == b`, `ne(a, b)` équivaut à `a != b`, `gt(a, b)` équivaut à `a > b` et `ge(a, b)` équivaut à `a >= b`. Notez que ces fonctions peuvent renvoyer n'importe quelle valeur, convertible ou non en booléen. Voir comparisons pour plus d'informations sur les méthodes de comparaison riches.

En général, les opérations logiques s'appliquent aussi à tous les objets et implémentent les tests de vérité, d'identité et les opérations booléennes :

operator.**not_**(*obj*)

operator.**__not__**(*obj*)

Renvoie le résultat de `not` *obj*. (Notez qu'il n'existe pas de méthode `__not__()` pour les instances d'objet ; seul le cœur de l'interpréteur définit cette opération. Le résultat dépend des méthodes `__bool__()` et `__len__()`.)

operator.**truth**(*obj*)

Renvoie `True` si *obj* est vrai, et `False` dans le cas contraire. Équivaut à utiliser le constructeur de `bool`.

operator.**is_**(*a*, *b*)

Renvoie `a is b`. Vérifie si les deux paramètres sont le même objet.

operator.**is_not**(*a*, *b*)

Renvoie `a is not b`. Vérifie si les deux paramètres sont deux objets distincts.

operator.**is_none**(*a*)

Return `a is None`. Tests object identity.

Ajouté dans la version 3.14.

operator.**is_not_none**(*a*)

Return `a is not None`. Tests object identity.

Ajouté dans la version 3.14.

Les opérations mathématiques ou bit à bit sont les plus nombreuses :

operator.**abs**(*obj*)

operator.**__abs__**(*obj*)

Renvoie la valeur absolue de *obj*.

operator.**add**(*a*, *b*)

operator.**__add__**(*a*, *b*)

Renvoie `a + b` où *a* et *b* sont des nombres.

operator.**and_**(*a*, *b*)

operator.**__and__**(*a*, *b*)

Renvoie le *et* bit à bit de *a* et *b*.

operator.**floordiv**(*a*, *b*)

operator.**__floordiv__**(*a*, *b*)

Renvoie `a // b`.

operator.**index**(*a*)

operator.**__index__**(*a*)

Renvoie *a* converti en entier. Équivaut à `a.__index__()`.

Modifié dans la version 3.10 : Le résultat a toujours le type `int`. Précédemment, le résultat aurait pu être une instance d'une sous-classe de `int`.

operator.**inv**(*obj*)

operator.**invert**(*obj*)

operator.**__inv__**(*obj*)

operator.**__invert__**(*obj*)

Renvoie l'inverse bit à bit du nombre *obj*. Équivaut à `~obj`.

operator.**lshift**(*a*, *b*)

operator.**__lshift__**(*a*, *b*)

Renvoie le décalage de *b* bits vers la gauche de *a*.

operator.**mod**(*a*, *b*)

operator.**__mod__**(*a*, *b*)

Renvoie `a % b`.

operator.**mul**(*a*, *b*)

operator.**__mul__**(*a*, *b*)

Renvoie `a * b` où *a* et *b* sont des nombres.

operator.**matmul**(*a*, *b*)

operator.**__matmul__**(*a*, *b*)

Renvoie `a @ b`.

Ajouté dans la version 3.5.

operator.**neg**(*obj*)

operator.**__neg__**(*obj*)

Renvoie l'opposé de *obj* (`-obj`).

operator.**or_**(*a*, *b*)

operator.**__or__**(*a*, *b*)

Renvoie le *ou* bit à bit de *a* et *b*.

operator.**pos**(*obj*)

operator.**__pos__**(*obj*)

Renvoie la valeur positive de *obj* (`+obj`).

operator.**pow**(*a*, *b*)

operator.**__pow__**(*a*, *b*)

Renvoie `a ** b` où *a* et *b* sont des nombres.

operator.**rshift**(*a*, *b*)

operator.**__rshift__**(*a*, *b*)

Renvoie le décalage de *b* bits vers la droite de *a*.

operator.**sub**(*a*, *b*)

operator.**__sub__**(*a*, *b*)

Renvoie `a - b`.

operator.**truediv**(*a*, *b*)

operator.**__truediv__**(*a*, *b*)

Renvoie `a / b` où 2/3 est 0.66 et non 0. Appelée aussi division « réelle ».

operator.**xor**(*a*, *b*)

operator.**__xor__**(*a*, *b*)

Renvoie le ou exclusif bit à bit de *a* et *b*.

Les opérations sur séquences (et pour certaines, sur correspondances) sont :

operator.**concat**(*a*, *b*)

operator.**__concat__**(*a*, *b*)

Renvoie `a + b` où *a* et *b* sont des séquences.

operator.**contains**(*a*, *b*)

operator.**__contains__**(*a*, *b*)

Renvoie le résultat du test `b in a`. Notez que les opérandes sont inversées.

operator.**countOf**(*a*, *b*)

Renvoie le nombre d'occurrences de *b* dans *a*.

operator.**delitem**(*a*, *b*)

operator.**__delitem__**(*a*, *b*)

Renvoie la valeur de *a* à l'indice *b*.

operator.**getitem**(*a*, *b*)

operator.**__getitem__**(*a*, *b*)

Renvoie la valeur de *a* à l'indice *b*.

operator.**indexOf**(*a*, *b*)

Renvoie l'indice de la première occurrence de *b* dans *a*.

operator.**setitem**(*a*, *b*, *c*)

operator.**__setitem__**(*a*, *b*, *c*)

Affecte *c* dans *a* à l'indice *b*.

operator.**length_hint**(*obj*, *default=0*)

Renvoie une estimation de la taille de l'objet *obj*. Tente d'abord de renvoyer la taille réelle, puis une estimation en appelant `object.__length_hint__()`, ou sinon la valeur par défaut.

Ajouté dans la version 3.4.

L'opération suivante marche avec les objets appelables :

operator.**call**(*obj*, */*, **args*, ***kwargs*)

operator.**__call__**(*obj*, */*, **args*, ***kwargs*)

Return `obj(*args, **kwargs)`.

Ajouté dans la version 3.11.

Le module *operator* définit aussi des fonctions pour la recherche générique d'attributs ou d'objets. Elles sont particulièrement utiles pour construire rapidement des accesseurs d'attributs à passer en paramètre à *map()*, *sorted()*, *itertools.groupby()* ou à toute autre fonction prenant une fonction en paramètre.

operator.**attrgetter**(*attr*)

operator.**attrgetter**(**attrs*)

Renvoie un objet appelable qui récupère *attr* de son opérande. Si plus d'un attribut est demandé, renvoie un *n*-uplet d'attributs. Les noms des attributs peuvent aussi comporter des points. Par exemple :

— Avec `f = attrgetter('name')`, l'appel `f(b)` renvoie `b.name`.
— Avec `f = attrgetter('name', 'date')`, l'appel `f(b)` renvoie `(b.name, b.date)`.
— Après `f = attrgetter('name.first', 'name.last')`, l'appel `f(b)` renvoie `(b.name.first, b.name.last)`.

Équivalent à :

```
def attrgetter(*items):
    if any(not isinstance(item, str) for item in items):
        raise TypeError('attribute name must be a string')
    if len(items) == 1:
        attr = items[0]
        def g(obj):
            return resolve_attr(obj, attr)
    else:
        def g(obj):
            return tuple(resolve_attr(obj, attr) for attr in items)
    return g
```

(suite sur la page suivante)

(suite de la page précédente)

```
def resolve_attr(obj, attr):
    for name in attr.split("."):
        obj = getattr(obj, name)
    return obj
```

operator.**itemgetter**(*item*)

operator.**itemgetter**(**items*)

Renvoie un objet appelable qui récupère *item* de l'opérande en utilisant la méthode `__getitem__()`. Si plusieurs *item* sont passés en paramètre, renvoie un *n*-uplet des valeurs récupérées. Par exemple :

— Avec `f = itemgetter(2)`, `f(r)` renvoie `r[2]`.

— Avec `g = itemgetter(2, 5, 3)`, `g(r)` renvoie `(r[2], r[5], r[3])`.

Équivalent à :

```
def itemgetter(*items):
    if len(items) == 1:
        item = items[0]
        def g(obj):
            return obj[item]
    else:
        def g(obj):
            return tuple(obj[item] for item in items)
    return g
```

Les *items* en entrée peuvent être de n'importe quel type tant que celui-ci est géré par la méthode `__getitem__()` de l'opérande. Les dictionnaires acceptent toute valeur *hachable*. Les listes, *n*-uplets et chaînes de caractères acceptent un index ou une tranche :

```
>>> itemgetter(1)('ABCDEFG')
'B'
>>> itemgetter(1, 3, 5)('ABCDEFG')
('B', 'D', 'F')
>>> itemgetter(slice(2, None))('ABCDEFG')
'CDEFG'
>>> soldier = dict(rank='captain', name='dotterbart')
>>> itemgetter('rank')(soldier)
'captain'
```

Exemple d'utilisation de `itemgetter()` pour récupérer des champs spécifiques d'un *n*-uplet :

```
>>> inventory = [('apple', 3), ('banana', 2), ('pear', 5), ('orange', 1)]
>>> getcount = itemgetter(1)
>>> list(map(getcount, inventory))
[3, 2, 5, 1]
>>> sorted(inventory, key=getcount)
[('orange', 1), ('banana', 2), ('apple', 3), ('pear', 5)]
```

operator.**methodcaller**(*name*, /, **args*, ***kwargs*)

Renvoie un objet appelable qui appelle la méthode *name* de son opérande. Si des paramètres supplémentaires et/ou des paramètres nommés sont donnés, ils seront aussi passés à la méthode. Par exemple :

— Avec `f = methodcaller('name')`, `f(b)` renvoie `b.name()`.

— Avec `f = methodcaller('name', 'foo', bar=1)`, `f(b)` renvoie `b.name('foo', bar=1)`.

Équivalent à :

```
def methodcaller(name, /, *args, **kwargs):
    def caller(obj):
```

(suite sur la page suivante)

(suite de la page précédente)

```
        return getattr(obj, name)(*args, **kwargs)
    return caller
```

10.3.1 Correspondances entre opérateurs et fonctions

Le tableau montre la correspondance entre les symboles des opérateurs Python et les fonctions du module `operator`.

Opération	Syntaxe	Fonction
Addition	`a + b`	`add(a, b)`
Concaténation	`seq1 + seq2`	`concat(seq1, seq2)`
Test d'inclusion	`obj in seq`	`contains(seq, obj)`
Division	`a / b`	`truediv(a, b)`
Division	`a // b`	`floordiv(a, b)`
Et bit à bit	`a & b`	`and_(a, b)`
Ou exclusif bit à bit	`a ^ b`	`xor(a, b)`
Inversion bit à bit	`~ a`	`invert(a)`
Ou bit à bit	`a | b`	`or_(a, b)`
Exponentiation	`a ** b`	`pow(a, b)`
Identité	`a is b`	`is_(a, b)`
Identité	`a is not b`	`is_not(a, b)`
Identité	`a is None`	`is_none(a)`
Identité	`a is not None`	`is_not_none(a)`
Affectation par index	`obj[k] = v`	`setitem(obj, k, v)`
Suppression par index	`del obj[k]`	`delitem(obj, k)`
Indexation	`obj[k]`	`getitem(obj, k)`
Décalage bit à bit gauche	`a << b`	`lshift(a, b)`
Modulo	`a % b`	`mod(a, b)`
Multiplication	`a * b`	`mul(a, b)`
Multiplication matricielle	`a @ b`	`matmul(a, b)`
Opposé	`- a`	`neg(a)`
Négation (logique)	`not a`	`not_(a)`
Valeur positive	`+ a`	`pos(a)`
Décalage bit à bit droite	`a >> b`	`rshift(a, b)`
Affectation par tranche	`seq[i:j] = values`	`setitem(seq, slice(i, j), values)`
Suppression par tranche	`del seq[i:j]`	`delitem(seq, slice(i, j))`
Tranche	`seq[i:j]`	`getitem(seq, slice(i, j))`
Formatage de chaînes de caractères	`s % obj`	`mod(s, obj)`
Soustraction	`a - b`	`sub(a, b)`
Test de véracité	`obj`	`truth(obj)`
Ordre	`a < b`	`lt(a, b)`
Ordre	`a <= b`	`le(a, b)`
Égalité	`a == b`	`eq(a, b)`
Inégalité	`a != b`	`ne(a, b)`
Ordre	`a >= b`	`ge(a, b)`
Ordre	`a > b`	`gt(a, b)`

10.3.2 Opérateurs en-place

Beaucoup d'opérations ont une version travaillant « en-place ». Les fonctions listées ci-dessous fournissent un accès plus direct aux opérateurs en-place que la syntaxe Python habituelle ; par exemple, l'expression *statement* `x += y` équivaut à `x = operator.iadd(x, y)`. Autrement dit, l'expression `z = operator.iadd(x, y)` équivaut à l'expression composée `z = x; z += y`.

Dans ces exemples, notez que lorsqu'une méthode en-place est appelée, le calcul et l'affectation sont effectués en deux étapes distinctes. Les fonctions en-place de la liste ci-dessous ne font que la première, en appelant la méthode

en-place. La seconde étape, l'affectation, n'est pas effectuée.

Pour des paramètres non-mutables comme les chaînes de caractères, les nombres et les *n*-uplets, la nouvelle valeur est calculée, mais pas affectée à la variable d'entrée :

```
>>> a = 'hello'
>>> iadd(a, ' world')
'hello world'
>>> a
'hello'
```

Pour des paramètres mutables comme les listes et les dictionnaires, la méthode en-place modifiera la valeur, aucune affectation ultérieure n'est nécessaire :

```
>>> s = ['h', 'e', 'l', 'l', 'o']
>>> iadd(s, [' ', 'w', 'o', 'r', 'l', 'd'])
['h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd']
>>> s
['h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd']
```

`operator.`**`iadd`**(*a*, *b*)

`operator.`**`__iadd__`**(*a*, *b*)

`a = iadd(a, b)` équivaut à `a += b`.

`operator.`**`iand`**(*a*, *b*)

`operator.`**`__iand__`**(*a*, *b*)

`a = iand(a, b)` équivaut à `a &= b`.

`operator.`**`iconcat`**(*a*, *b*)

`operator.`**`__iconcat__`**(*a*, *b*)

`a = iconcat(a, b)` équivaut à `a += b` où *a* et *b* sont des séquences.

`operator.`**`ifloordiv`**(*a*, *b*)

`operator.`**`__ifloordiv__`**(*a*, *b*)

`a = ifloordiv(a, b)` équivaut à `a //= b`.

`operator.`**`ilshift`**(*a*, *b*)

`operator.`**`__ilshift__`**(*a*, *b*)

`a = ilshift(a, b)` équivaut à `a <<= b`.

`operator.`**`imod`**(*a*, *b*)

`operator.`**`__imod__`**(*a*, *b*)

`a = imod(a, b)` équivaut à `a %= b`.

`operator.`**`imul`**(*a*, *b*)

`operator.`**`__imul__`**(*a*, *b*)

`a = imul(a, b)` équivaut à `a *= b`.

`operator.`**`imatmul`**(*a*, *b*)

`operator.`**`__imatmul__`**(*a*, *b*)

`a = imatmul(a, b)` équivaut à `a @= b`.

Ajouté dans la version 3.5.

`operator.`**`ior`**(*a*, *b*)

`operator.`**`__ior__`**(*a*, *b*)

`a = ior(a, b)` équivaut à `a |= b`.

`operator.`**`ipow`**(*a*, *b*)

`operator.__ipow__(a, b)`

`a = ipow(a, b)` équivaut à `a **= b`.

`operator.irshift(a, b)`

`operator.__irshift__(a, b)`

`a = irshift(a, b)` équivaut à `a >>= b`.

`operator.isub(a, b)`

`operator.__isub__(a, b)`

`a = isub(a, b)` équivaut à `a -= b`.

`operator.itruediv(a, b)`

`operator.__itruediv__(a, b)`

`a = itruediv(a, b)` équivaut à `a /= b`.

`operator.ixor(a, b)`

`operator.__ixor__(a, b)`

`a = ixor(a, b)` équivaut à `a ^= b`.

CHAPITRE 11

Accès aux Fichiers et aux Dossiers

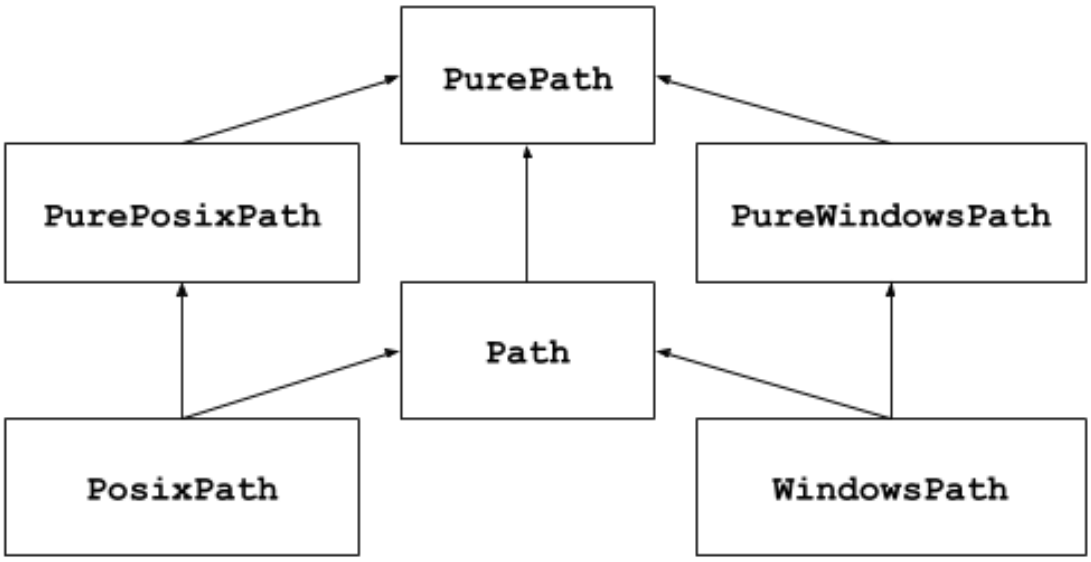
Les modules décrits dans ce chapitre servent à accéder aux fichiers et aux dossiers. Des modules, par exemple, pour lire les propriétés des fichiers, manipuler des chemins de manière portable, et créer des fichiers temporaires. La liste exhaustive des modules de ce chapitre est :

11.1 `pathlib` — Chemins de système de fichiers orientés objet

Ajouté dans la version 3.4.

Code source : Lib/pathlib/

Ce module offre des classes représentant le système de fichiers avec la sémantique appropriée pour différents systèmes d'exploitation. Les classes de chemins sont divisées en *chemins purs*, qui fournissent uniquement des opérations de manipulation sans entrées-sorties, et *chemins concrets*, qui héritent des chemins purs et fournissent également les opérations d'entrées-sorties.



Si vous n'avez jamais utilisé ce module précédemment, ou si vous n'êtes pas sûr de quelle classe est faite pour votre tâche, `Path` est très certainement ce dont vous avez besoin. Elle instancie un *chemin concret* pour la plateforme sur laquelle s'exécute le code.

Les chemins purs sont utiles dans certains cas particuliers ; par exemple :

1. Si vous voulez manipuler des chemins Windows sur une machine Unix (ou vice versa). Vous ne pouvez pas instancier un `WindowsPath` quand vous êtes sous Unix, mais vous pouvez instancier `PureWindowsPath`.
2. Vous voulez être sûr que votre code manipule des chemins sans réellement accéder au système d'exploitation. Dans ce cas, instancier une de ces classes pures peut être utile puisqu'elle ne possède tout simplement aucune opération permettant d'accéder au système d'exploitation.

Voir aussi

PEP 428 : le module *pathlib* – implémentation orientée-objet des chemins de systèmes de fichiers.

Voir aussi

Pour de la manipulation de chemins bas-niveau avec des chaînes de caractères, vous pouvez aussi utiliser le module `os.path`.

11.1.1 Utilisation basique

Importer la classe principale :

```
>>> from pathlib import Path
```

Lister les sous-dossiers :

```
>>> p = Path('.')
>>> [x for x in p.iterdir() if x.is_dir()]
[PosixPath('.hg'), PosixPath('docs'), PosixPath('dist'),
 PosixPath('__pycache__'), PosixPath('build')]
```

Lister les fichiers source Python dans cette arborescence de dossiers :

```
>>> list(p.glob('**/*.py'))
[PosixPath('test_pathlib.py'), PosixPath('setup.py'),
 PosixPath('pathlib.py'), PosixPath('docs/conf.py'),
 PosixPath('build/lib/pathlib.py')]
```

Naviguer à l'intérieur d'une arborescence de dossiers :

```
>>> p = Path('/etc')
>>> q = p / 'init.d' / 'reboot'
>>> q
PosixPath('/etc/init.d/reboot')
>>> q.resolve()
PosixPath('/etc/rc.d/init.d/halt')
```

Récupérer les propriétés de chemin :

```
>>> q.exists()
True
>>> q.is_dir()
False
```

Ouvrir un fichier :

```
>>> with q.open() as f: f.readline()
...
'#!/bin/bash\n'
```

11.1.2 Exceptions

exception pathlib.**UnsupportedOperation**

Une exception qui hérite de `NotImplementedError` levée quand une opération non prise en charge est appelée sur un objet *path*

Ajouté dans la version 3.13.

11.1.3 Chemins purs

Les objets chemins purs fournissent les opérations de gestion de chemin qui n'accèdent pas réellement au système de fichiers. Il y a trois façons d'accéder à ces classes que nous appelons aussi *familles* :

class pathlib.**PurePath**(**pathsegments*)

Une classe générique qui représente la famille de chemin du système (l'instancier crée soit un `PurePosixPath` soit un `PureWindowsPath`) :

```
>>> PurePath('setup.py')      # Running on a Unix machine
PurePosixPath('setup.py')
```

Each element of *pathsegments* can be either a string representing a path segment, or an object implementing the `os.PathLike` interface where the `__fspath__()` method returns a string, such as another path object :

```
>>> PurePath('foo', 'some/path', 'bar')
PurePosixPath('foo/some/path/bar')
>>> PurePath(Path('foo'), Path('bar'))
PurePosixPath('foo/bar')
```

Quand *pathsegments* est vide, le dossier courant est utilisé :

```
>>> PurePath()
PurePosixPath('.')
```

Si un segment est un chemin absolu, tous les segments précédents sont ignorés (comme `os.path.join()`) :

```
>>> PurePath('/etc', '/usr', 'lib64')
PurePosixPath('/usr/lib64')
>>> PureWindowsPath('c:/Windows', 'd:bar')
PureWindowsPath('d:bar')
```

Sous Windows, le nom de lecteur est conservé quand un chemin relatif à la racine (par exemple `r'\foo'`) est rencontré :

```
>>> PureWindowsPath('c:/Windows', '/Program Files')
PureWindowsPath('c:/Program Files')
```

Les points et slashs superflus sont supprimés, mais les doubles points (`'..'`) et les double slashes (`'//'`) en début de segment ne le sont pas, puisque cela changerait la signification du chemin pour différentes raisons (par exemple pour des liens symboliques ou des chemins UNC) :

```
>>> PurePath('foo//bar')
PurePosixPath('foo/bar')
>>> PurePath('//foo/bar')
PurePosixPath('//foo/bar')
>>> PurePath('foo/./bar')
PurePosixPath('foo/bar')
>>> PurePath('foo/../bar')
PurePosixPath('foo/../bar')
```

(une analyse naïve considérerait `PurePosixPath('foo/../bar')` équivalent à `PurePosixPath('bar')`, ce qui est faux si `foo` est un lien symbolique vers un autre dossier)

Les objets chemins purs implémentent l'interface `os.PathLike`, leur permettant d'être utilisés n'importe où l'interface est acceptée.

Modifié dans la version 3.6 : ajout de la gestion de l'interface `os.PathLike`.

class pathlib.**PurePosixPath**(**pathsegments*)

Sous-classe de `PurePath`, cette famille de chemin représente les chemins de systèmes de fichiers en dehors des chemins Windows :

```
>>> PurePosixPath('/etc/hosts')
PurePosixPath('/etc/hosts')
```

pathsegments est spécifié de manière similaire à `PurePath`.

class pathlib.**PureWindowsPath**(**pathsegments*)

Sous-classe de `PurePath`, cette famille de chemin représente les chemins de systèmes de fichiers Windows, y compris les chemins UNC (voir UNC paths) :

```
>>> PureWindowsPath('c:/', 'Users', 'Ximénez')
PureWindowsPath('c:/Users/Ximénez')
>>> PureWindowsPath('//server/share/file')
PureWindowsPath('//server/share/file')
```

pathsegments est spécifié de manière similaire à `PurePath`.

Sans tenir compte du système sur lequel vous êtes, vous pouvez instancier toutes ces classes, puisqu'elles ne fournissent aucune opération qui appelle le système d'exploitation.

Propriétés générales

Paths are immutable and *hashable*. Paths of a same flavour are comparable and orderable. These properties respect the flavour's case-folding semantics :

```
>>> PurePosixPath('foo') == PurePosixPath('FOO')
False
>>> PureWindowsPath('foo') == PureWindowsPath('FOO')
True
>>> PureWindowsPath('FOO') in { PureWindowsPath('foo') }
True
>>> PureWindowsPath('C:') < PureWindowsPath('d:')
True
```

Les chemins de différentes familles ne sont pas égaux et ne peuvent être ordonnés :

```
>>> PureWindowsPath('foo') == PurePosixPath('foo')
False
>>> PureWindowsPath('foo') < PurePosixPath('foo')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'PureWindowsPath' and
↪'PurePosixPath'
```

Opérateurs

L'opérateur slash aide à créer les chemins enfants, de manière similaire à `os.path.join()`. Si l'argument est un chemin absolu, le chemin précédent est ignoré. Sous Windows, le lecteur n'est pas effacé quand l'argument est un chemin relatif enraciné (par exemple `r'\foo'`) :

```
>>> p = PurePath('/etc')
>>> p
PurePosixPath('/etc')
>>> p / 'init.d' / 'apache2'
PurePosixPath('/etc/init.d/apache2')
>>> q = PurePath('bin')
>>> '/usr' / q
PurePosixPath('/usr/bin')
>>> p / '/an_absolute_path'
PurePosixPath('/an_absolute_path')
>>> PureWindowsPath('c:/Windows', '/Program Files')
PureWindowsPath('c:/Program Files')
```

Un objet chemin peut être utilisé n'importe où un objet implémentant `os.PathLike` est accepté :

```
>>> import os
>>> p = PurePath('/etc')
>>> os.fspath(p)
'/etc'
```

La représentation d'un chemin en chaîne de caractères est celle du chemin brut du système de fichiers lui-même (dans sa forme native, c.-à-d. avec des antislashs sous Windows), telle que vous pouvez la passer à n'importe quelle fonction prenant un chemin en tant que chaîne de caractères :

```
>>> p = PurePath('/etc')
>>> str(p)
'/etc'
>>> p = PureWindowsPath('c:/Program Files')
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> str(p)
'c:\\Program Files'
```

De manière similaire, appeler `bytes` sur un chemin donne le chemin brut du système de fichiers en tant que bytes, tel qu'encodé par `os.fsencode()` :

```
>>> bytes(p)
b'/etc'
```

> **Note**
>
> Appeler `bytes` est seulement recommandé sous Unix. Sous Windows, la forme Unicode est la représentation canonique des chemins du système de fichiers.

Accéder aux parties individuelles

Pour accéder aux parties individuelles (composantes) d'un chemin, utilisez les propriétés suivantes :

`PurePath.`**`parts`**

Un tuple donnant accès aux différentes composantes du chemin :

```
>>> p = PurePath('/usr/bin/python3')
>>> p.parts
('/', 'usr', 'bin', 'python3')

>>> p = PureWindowsPath('c:/Program Files/PSF')
>>> p.parts
('c:\\', 'Program Files', 'PSF')
```

(notez comme le lecteur et la racine locale sont regroupés en une seule partie)

Méthodes et propriétés

Les chemins purs fournissent les méthodes et propriétés suivantes :

`PurePath.`**`parser`**

The implementation of the `os.path` module used for low-level path parsing and joining : either `posixpath` or `ntpath`.

Ajouté dans la version 3.13.

`PurePath.`**`drive`**

Une chaîne représentant la lettre du lecteur ou le nom, s'il y en a un :

```
>>> PureWindowsPath('c:/Program Files/').drive
'c:'
>>> PureWindowsPath('/Program Files/').drive
''
>>> PurePosixPath('/etc').drive
''
```

Les partages UNC sont aussi considérés comme des lecteurs :

```
>>> PureWindowsPath('//host/share/foo.txt').drive
'\\\\host\\share'
```

`PurePath.`**`root`**

Une chaîne de caractères représentant la racine (locale ou globale), s'il y en a une :

```
>>> PureWindowsPath('c:/Program Files/').root
'\\'
>>> PureWindowsPath('c:Program Files/').root
''
>>> PurePosixPath('/etc').root
'/'
```

Les partages UNC ont toujours une racine :

```
>>> PureWindowsPath('//host/share').root
'\\'
```

Si le chemin commence par plus de deux slashes successifs, `PurePosixPath` n'en conserve qu'un :

```
>>> PurePosixPath('//etc').root
'//'
>>> PurePosixPath('///etc').root
'/'
>>> PurePosixPath('////etc').root
'/'
```

Note

Ce comportement se conforme au paragraphe 4.11 Pathname Resolution des *Open Group Base Specifications* version 6 :

"A pathname that begins with two successive slashes may be interpreted in an implementation-defined manner, although more than two leading slashes shall be treated as a single slash."

`PurePath.`**`anchor`**

La concaténation du lecteur et de la racine :

```
>>> PureWindowsPath('c:/Program Files/').anchor
'c:\\'
>>> PureWindowsPath('c:Program Files/').anchor
'c:'
>>> PurePosixPath('/etc').anchor
'/'
>>> PureWindowsPath('//host/share').anchor
'\\\\host\\share\\'
```

`PurePath.`**`parents`**

Une séquence immuable fournissant accès aux ancêtres logiques du chemin :

```
>>> p = PureWindowsPath('c:/foo/bar/setup.py')
>>> p.parents[0]
PureWindowsPath('c:/foo/bar')
>>> p.parents[1]
PureWindowsPath('c:/foo')
>>> p.parents[2]
PureWindowsPath('c:/')
```

Modifié dans la version 3.10 : La séquence de *parents* prend désormais en charge les *tranches* et les valeurs d'index négatives.

`PurePath.`**`parent`**

Le parent logique du chemin :

```
>>> p = PurePosixPath('/a/b/c/d')
>>> p.parent
PurePosixPath('/a/b/c')
```

Vous ne pouvez pas aller au-delà d'une ancre, ou d'un chemin vide :

```
>>> p = PurePosixPath('/')
>>> p.parent
PurePosixPath('/')
>>> p = PurePosixPath('.')
>>> p.parent
PurePosixPath('.')
```

Note

C'est une opération purement lexicale, d'où le comportement suivant :

```
>>> p = PurePosixPath('foo/..')
>>> p.parent
PurePosixPath('foo')
```

Si vous voulez remonter un chemin arbitraire du système de fichiers, il est recommandé d'appeler d'abord `Path.resolve()` de manière à résoudre les liens symboliques et éliminer les composantes `".."`.

`PurePath.`**`name`**

Une chaîne représentant la composante finale du chemin, en excluant le lecteur et la racine, si présent :

```
>>> PurePosixPath('my/library/setup.py').name
'setup.py'
```

Les noms de lecteur UNC ne sont pas pris en compte :

```
>>> PureWindowsPath('//some/share/setup.py').name
'setup.py'
>>> PureWindowsPath('//some/share').name
''
```

`PurePath.`**`suffix`**

The last dot-separated portion of the final component, if any :

```
>>> PurePosixPath('my/library/setup.py').suffix
'.py'
>>> PurePosixPath('my/library.tar.gz').suffix
'.gz'
>>> PurePosixPath('my/library').suffix
''
```

Ceci est communément appelé une extension de fichier.

Modifié dans la version 3.14 : A single dot (".") is considered a valid suffix.

`PurePath.`**`suffixes`**

Une liste de suffixes du chemin concerné, souvent appelées extensions de fichier :

```
>>> PurePosixPath('my/library.tar.gar').suffixes
['.tar', '.gar']
>>> PurePosixPath('my/library.tar.gz').suffixes
```

(suite sur la page suivante)

(suite de la page précédente)

```
['.tar', '.gz']
>>> PurePosixPath('my/library').suffixes
[]
```

Modifié dans la version 3.14 : A single dot (".") is considered a valid suffix.

`PurePath.stem`

La composante finale du chemin, sans son suffixe :

```
>>> PurePosixPath('my/library.tar.gz').stem
'library.tar'
>>> PurePosixPath('my/library.tar').stem
'library'
>>> PurePosixPath('my/library').stem
'library'
```

`PurePath.as_posix()`

Renvoie une représentation en chaîne de caractères du chemin avec des slashs (/) :

```
>>> p = PureWindowsPath('c:\\windows')
>>> str(p)
'c:\\windows'
>>> p.as_posix()
'c:/windows'
```

`PurePath.is_absolute()`

Renvoie si le chemin est absolu ou non. Un chemin est considéré absolu s'il a une racine et un lecteur (si la famille le permet) :

```
>>> PurePosixPath('/a/b').is_absolute()
True
>>> PurePosixPath('a/b').is_absolute()
False

>>> PureWindowsPath('c:/a/b').is_absolute()
True
>>> PureWindowsPath('/a/b').is_absolute()
False
>>> PureWindowsPath('c:').is_absolute()
False
>>> PureWindowsPath('//some/share').is_absolute()
True
```

`PurePath.is_relative_to(other)`

Renvoie si ce chemin est relatif ou non au chemin *other*.

```
>>> p = PurePath('/etc/passwd')
>>> p.is_relative_to('/etc')
True
>>> p.is_relative_to('/usr')
False
```

This method is string-based ; it neither accesses the filesystem nor treats "..." segments specially. The following code is equivalent :

```
>>> u = PurePath('/usr')
>>> u == p or u in p.parents
False
```

Ajouté dans la version 3.9.

Deprecated since version 3.12, removed in version 3.14 : Passing additional arguments is deprecated ; if supplied, they are joined with *other*.

`PurePath.is_reserved()`

Avec `PureWindowsPath`, renvoie `True` si le chemin est considéré réservé sous Windows, `False` sinon. Avec `PurePosixPath`, `False` est systématiquement renvoyé.

Modifié dans la version 3.13 : Windows path names that contain a colon, or end with a dot or a space, are considered reserved. UNC paths may be reserved.

Deprecated since version 3.13, will be removed in version 3.15 : This method is deprecated ; use `os.path.isreserved()` to detect reserved paths on Windows.

`PurePath.joinpath(*pathsegments)`

Calling this method is equivalent to combining the path with each of the given *pathsegments* in turn :

```
>>> PurePosixPath('/etc').joinpath('passwd')
PurePosixPath('/etc/passwd')
>>> PurePosixPath('/etc').joinpath(PurePosixPath('passwd'))
PurePosixPath('/etc/passwd')
>>> PurePosixPath('/etc').joinpath('init.d', 'apache2')
PurePosixPath('/etc/init.d/apache2')
>>> PureWindowsPath('c:').joinpath('/Program Files')
PureWindowsPath('c:/Program Files')
```

`PurePath.full_match(pattern, *, case_sensitive=None)`

Match this path against the provided glob-style pattern. Return `True` if matching is successful, `False` otherwise. For example :

```
>>> PurePath('a/b.py').full_match('a/*.py')
True
>>> PurePath('a/b.py').full_match('*.py')
False
>>> PurePath('/a/b/c.py').full_match('/a/**')
True
>>> PurePath('/a/b/c.py').full_match('**/*.py')
True
```

Voir aussi

documentation *Pattern language*.

Comme pour les autres méthodes, la sensibilité à la casse est la sensibilité par défaut de la plate-forme :

```
>>> PurePosixPath('b.py').full_match('*.PY')
False
>>> PureWindowsPath('b.py').full_match('*.PY')
True
```

La sensibilité à la casse peut être choisie en passant `True` ou `False` à l'argument *case_sensitive*.

Ajouté dans la version 3.13.

`PurePath.match(pattern, *, case_sensitive=None)`

Match this path against the provided non-recursive glob-style pattern. Return `True` if matching is successful, `False` otherwise.

This method is similar to `full_match()`, but empty patterns aren't allowed (`ValueError` is raised), the recursive wildcard "`**`" isn't supported (it acts like non-recursive "`*`"), and if a relative pattern is provided, then matching is done from the right :

```
>>> PurePath('a/b.py').match('*.py')
True
>>> PurePath('/a/b/c.py').match('b/*.py')
True
>>> PurePath('/a/b/c.py').match('a/*.py')
False
```

Modifié dans la version 3.12 : Le paramètre *pattern* accepte un *path-like object*.

Modifié dans la version 3.12 : ajout du paramètre *case_sensitive*

`PurePath.relative_to(other, walk_up=False)`

Compute a version of this path relative to the path represented by *other*. If it's impossible, `ValueError` is raised :

```
>>> p = PurePosixPath('/etc/passwd')
>>> p.relative_to('/')
PurePosixPath('etc/passwd')
>>> p.relative_to('/etc')
PurePosixPath('passwd')
>>> p.relative_to('/usr')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "pathlib.py", line 941, in relative_to
    raise ValueError(error_message.format(str(self), str(formatted)))
ValueError: '/etc/passwd' is not in the subpath of '/usr' OR one path is␣
↪relative and the other is absolute.
```

When *walk_up* is false (the default), the path must start with *other*. When the argument is true, `..` entries may be added to form the relative path. In all other cases, such as the paths referencing different drives, `ValueError` is raised. :

```
>>> p.relative_to('/usr', walk_up=True)
PurePosixPath('../etc/passwd')
>>> p.relative_to('foo', walk_up=True)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "pathlib.py", line 941, in relative_to
    raise ValueError(error_message.format(str(self), str(formatted)))
ValueError: '/etc/passwd' is not on the same drive as 'foo' OR one path is␣
↪relative and the other is absolute.
```

Avertissement

This function is part of `PurePath` and works with strings. It does not check or access the underlying file structure. This can impact the *walk_up* option as it assumes that no symlinks are present in the path ; call `resolve()` first if necessary to resolve symlinks.

Modifié dans la version 3.12 : The *walk_up* parameter was added (old behavior is the same as `walk_up=False`).

Deprecated since version 3.12, removed in version 3.14 : Passing additional positional arguments is deprecated ; if supplied, they are joined with *other*.

`PurePath.with_name(name)`

Renvoie un nouveau chemin avec `name` changé. Si le chemin original n'a pas de nom, une `ValueError` est levée :

```
>>> p = PureWindowsPath('c:/Downloads/pathlib.tar.gz')
>>> p.with_name('setup.py')
PureWindowsPath('c:/Downloads/setup.py')
>>> p = PureWindowsPath('c:/')
>>> p.with_name('setup.py')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/antoine/cpython/default/Lib/pathlib.py", line 751, in with_name
    raise ValueError("%r has an empty name" % (self,))
ValueError: PureWindowsPath('c:/') has an empty name
```

`PurePath.with_stem(stem)`

Renvoie un nouveau chemin avec `stem` changé. Si le chemin original n'a pas de nom, *ValueError* est levée :

```
>>> p = PureWindowsPath('c:/Downloads/draft.txt')
>>> p.with_stem('final')
PureWindowsPath('c:/Downloads/final.txt')
>>> p = PureWindowsPath('c:/Downloads/pathlib.tar.gz')
>>> p.with_stem('lib')
PureWindowsPath('c:/Downloads/lib.gz')
>>> p = PureWindowsPath('c:/')
>>> p.with_stem('')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/antoine/cpython/default/Lib/pathlib.py", line 861, in with_stem
    return self.with_name(stem + self.suffix)
  File "/home/antoine/cpython/default/Lib/pathlib.py", line 851, in with_name
    raise ValueError("%r has an empty name" % (self,))
ValueError: PureWindowsPath('c:/') has an empty name
```

Ajouté dans la version 3.9.

`PurePath.with_suffix(suffix)`

Renvoie un nouveau chemin avec `suffix` changé. Si le chemin original n'a pas de suffixe, le nouveau *suffix* est alors ajouté. Si *suffix* est une chaîne vide, le suffixe d'origine est retiré :

```
>>> p = PureWindowsPath('c:/Downloads/pathlib.tar.gz')
>>> p.with_suffix('.bz2')
PureWindowsPath('c:/Downloads/pathlib.tar.bz2')
>>> p = PureWindowsPath('README')
>>> p.with_suffix('.txt')
PureWindowsPath('README.txt')
>>> p = PureWindowsPath('README.txt')
>>> p.with_suffix('')
PureWindowsPath('README')
```

Modifié dans la version 3.14 : A single dot (".") is considered a valid suffix. In previous versions, `ValueError` is raised if a single dot is supplied.

`PurePath.with_segments(*pathsegments)`

Create a new path object of the same type by combining the given *pathsegments*. This method is called whenever a derivative path is created, such as from `parent` and `relative_to()`. Subclasses may override this method to pass information to derivative paths, for example :

```
from pathlib import PurePosixPath

class MyPath(PurePosixPath):
    def __init__(self, *pathsegments, session_id):
        super().__init__(*pathsegments)
        self.session_id = session_id

    def with_segments(self, *pathsegments):
        return type(self)(*pathsegments, session_id=self.session_id)

etc = MyPath('/etc', session_id=42)
hosts = etc / 'hosts'
print(hosts.session_id)  # 42
```

Ajouté dans la version 3.12.

11.1.4 Chemins concrets

Les chemins concrets sont des sous-classes des chemins purs. En plus des opérations fournies par ces derniers, ils fournissent aussi des méthodes pour faire appel au système sur des objets chemin. Il y a trois façons d'instancier des chemins concrets :

class pathlib.**Path**(**pathsegments*)

Une sous-classe de `PurePath`, cette classe représente les chemins concrets d'une famille de chemins de système de fichiers (l'instancier créé soit un `PosixPath`, soit un `WindowsPath`) :

```
>>> Path('setup.py')
PosixPath('setup.py')
```

pathsegments est spécifié de manière similaire à `PurePath`.

class pathlib.**PosixPath**(**pathsegments*)

Une sous-classe de `Path` et `PurePosixPath`, cette classe représente les chemins concrets de systèmes de fichiers non Windows :

```
>>> PosixPath('/etc/hosts')
PosixPath('/etc/hosts')
```

pathsegments est spécifié de manière similaire à `PurePath`.

Modifié dans la version 3.13 : Raises `UnsupportedOperation` on Windows. In previous versions, `NotImplementedError` was raised instead.

class pathlib.**WindowsPath**(**pathsegments*)

Une sous-classe de `Path` et `PureWindowsPath`, cette classe représente les chemins concrets de systèmes de fichiers Windows :

```
>>> WindowsPath('c:/', 'Users', 'Ximénez')
WindowsPath('c:/Users/Ximénez')
```

pathsegments est spécifié de manière similaire à `PurePath`.

Modifié dans la version 3.13 : Raises `UnsupportedOperation` on non-Windows platforms. In previous versions, `NotImplementedError` was raised instead.

Vous pouvez seulement instancier la classe de la famille qui correspond à votre système (permettre des appels au système pour des familles de chemins non compatibles pourrait mener à des bogues ou à des pannes de votre application) :

```
>>> import os
>>> os.name
'posix'
>>> Path('setup.py')
PosixPath('setup.py')
>>> PosixPath('setup.py')
PosixPath('setup.py')
>>> WindowsPath('setup.py')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "pathlib.py", line 798, in __new__
    % (cls.__name__,))
UnsupportedOperation: cannot instantiate 'WindowsPath' on your system
```

Some concrete path methods can raise an `OSError` if a system call fails (for example because the path doesn't exist).

Analyse et génération des URI

Concrete path objects can be created from, and represented as, 'file' URIs conforming to **RFC 8089**.

> **Note**
>
> File URIs are not portable across machines with different *filesystem encodings*.

classmethod `Path.`**`from_uri`**(*uri*)

Return a new path object from parsing a 'file' URI. For example :

```
>>> p = Path.from_uri('file:///etc/hosts')
PosixPath('/etc/hosts')
```

On Windows, DOS device and UNC paths may be parsed from URIs :

```
>>> p = Path.from_uri('file:///c:/windows')
WindowsPath('c:/windows')
>>> p = Path.from_uri('file://server/share')
WindowsPath('//server/share')
```

Plusieurs variants du formulaire sont pris en charge :

```
>>> p = Path.from_uri('file:////server/share')
WindowsPath('//server/share')
>>> p = Path.from_uri('file://///server/share')
WindowsPath('//server/share')
>>> p = Path.from_uri('file:c:/windows')
WindowsPath('c:/windows')
>>> p = Path.from_uri('file:/c|/windows')
WindowsPath('c:/windows')
```

`ValueError` is raised if the URI does not start with `file:`, or the parsed path isn't absolute.

Ajouté dans la version 3.13.

`Path.`**`as_uri`**()

Represent the path as a 'file' URI. `ValueError` is raised if the path isn't absolute.

```
>>> p = PosixPath('/etc/passwd')
>>> p.as_uri()
```

(suite sur la page suivante)

(suite de la page précédente)

```
'file:///etc/passwd'
>>> p = WindowsPath('c:/Windows')
>>> p.as_uri()
'file:///c:/Windows'
```

For historical reasons, this method is also available from `PurePath` objects. However, its use of `os.fsencode()` makes it strictly impure.

Expansion et résolution de chemins

classmethod `Path.home()`

Renvoie un nouveau chemin représentant le dossier d'accueil de l'utilisateur (comme renvoyé par `os.path.expanduser()` avec la construction ~). Si le dossier d'accueil de l'utilisateur ne peut être résolu, `RuntimeError` est levée.

```
>>> Path.home()
PosixPath('/home/antoine')
```

Ajouté dans la version 3.5.

`Path.expanduser()`

Renvoie un nouveau chemin avec les résolutions des constructions ~ et `~user`, comme renvoyé par `os.path.expanduser()`. Si le dossier d'accueil de l'utilisateur ne peut être résolu, `RuntimeError` est levée.

```
>>> p = PosixPath('~/films/Monty Python')
>>> p.expanduser()
PosixPath('/home/eric/films/Monty Python')
```

Ajouté dans la version 3.5.

classmethod `Path.cwd()`

Renvoie un nouveau chemin représentant le dossier courant (comme renvoyé par `os.getcwd()`) :

```
>>> Path.cwd()
PosixPath('/home/antoine/pathlib')
```

`Path.absolute()`

Rend le chemin absolu, sans normaliser ou résoudre les liens symboliques. Renvoie un nouveau :

```
>>> p = Path('tests')
>>> p
PosixPath('tests')
>>> p.absolute()
PosixPath('/home/antoine/pathlib/tests')
```

`Path.resolve(`*strict=False*`)`

Rend le chemin absolu, résolvant les liens symboliques. Un nouveau chemin est renvoyé :

```
>>> p = Path()
>>> p
PosixPath('.')
>>> p.resolve()
PosixPath('/home/antoine/pathlib')
```

Les composantes "`..`" sont aussi éliminées (c'est la seule méthode pour le faire) :

```
>>> p = Path('docs/../setup.py')
>>> p.resolve()
PosixPath('/home/antoine/pathlib/setup.py')
```

If a path doesn't exist or a symlink loop is encountered, and *strict* is `True`, `OSError` is raised. If *strict* is `False`, the path is resolved as far as possible and any remainder is appended without checking whether it exists.

Modifié dans la version 3.6 : The *strict* parameter was added (pre-3.6 behavior is strict).

Modifié dans la version 3.13 : Symlink loops are treated like other errors : `OSError` is raised in strict mode, and no exception is raised in non-strict mode. In previous versions, `RuntimeError` is raised no matter the value of *strict*.

`Path.readlink()`

Renvoie le chemin vers lequel le lien symbolique pointe (tel que renvoyé par `os.readlink()`) :

```
>>> p = Path('mylink')
>>> p.symlink_to('setup.py')
>>> p.readlink()
PosixPath('setup.py')
```

Ajouté dans la version 3.9.

Modifié dans la version 3.13 : Raises `UnsupportedOperation` if `os.readlink()` is not available. In previous versions, `NotImplementedError` was raised.

Requête sur type et statut du fichier

Modifié dans la version 3.8 : `exists()`, `is_dir()`, `is_file()`, `is_mount()`, `is_symlink()`, `is_block_device()`, `is_char_device()`, `is_fifo()`, `is_socket()` now return `False` instead of raising an exception for paths that contain characters unrepresentable at the OS level.

Modifié dans la version 3.14 : The methods given above now return `False` instead of raising any `OSError` exception from the operating system. In previous versions, some kinds of `OSError` exception are raised, and others suppressed. The new behaviour is consistent with `os.path.exists()`, `os.path.isdir()`, etc. Use `stat()` to retrieve the file status without suppressing exceptions.

`Path.stat(*, follow_symlinks=True)`

Return an `os.stat_result` object containing information about this path, like `os.stat()`. The result is looked up at each call to this method.

Cette méthode suit normalement les liens symboliques ; pour appeler `stat` sur un lien symbolique lui-même, ajoutez l'argument `follow_symlinks=False`, ou utilisez `lstat()`.

```
>>> p = Path('setup.py')
>>> p.stat().st_size
956
>>> p.stat().st_mtime
1327883547.852554
```

Modifié dans la version 3.10 : ajout du paramètre *follow_symlinks*.

`Path.lstat()`

Comme `Path.stat()`, mais, si le chemin pointe vers un lien symbolique, renvoie les informations du lien symbolique plutôt que celui de sa cible.

`Path.exists(*, follow_symlinks=True)`

Return `True` if the path points to an existing file or directory. `False` will be returned if the path is invalid, inaccessible or missing. Use `Path.stat()` to distinguish between these cases.

This method normally follows symlinks ; to check if a symlink exists, add the argument `follow_symlinks=False`.

```
>>> Path('.').exists()
True
>>> Path('setup.py').exists()
True
>>> Path('/etc').exists()
True
>>> Path('nonexistentfile').exists()
False
```

Modifié dans la version 3.12 : ajout du paramètre *follow_symlinks*.

`Path.is_file(*, follow_symlinks=True)`

Return `True` if the path points to a regular file. `False` will be returned if the path is invalid, inaccessible or missing, or if it points to something other than a regular file. Use `Path.stat()` to distinguish between these cases.

This method normally follows symlinks ; to exclude symlinks, add the argument `follow_symlinks=False`.

Modifié dans la version 3.13 : ajout du paramètre *follow_symlinks*.

`Path.is_dir(*, follow_symlinks=True)`

Return `True` if the path points to a directory. `False` will be returned if the path is invalid, inaccessible or missing, or if it points to something other than a directory. Use `Path.stat()` to distinguish between these cases.

This method normally follows symlinks ; to exclude symlinks to directories, add the argument `follow_symlinks=False`.

Modifié dans la version 3.13 : ajout du paramètre *follow_symlinks*.

`Path.is_symlink()`

Return `True` if the path points to a symbolic link, even if that symlink is broken. `False` will be returned if the path is invalid, inaccessible or missing, or if it points to something other than a symbolic link. Use `Path.stat()` to distinguish between these cases.

`Path.is_junction()`

Return `True` if the path points to a junction, and `False` for any other type of file. Currently only Windows supports junctions.

Ajouté dans la version 3.12.

`Path.is_mount()`

Return `True` if the path is a *mount point* : a point in a file system where a different file system has been mounted. On POSIX, the function checks whether *path*'s parent, `path/..`, is on a different device than *path*, or whether `path/..` and *path* point to the same i-node on the same device --- this should detect mount points for all Unix and POSIX variants. On Windows, a mount point is considered to be a drive letter root (e.g. `c:\`), a UNC share (e.g. `\\server\share`), or a mounted filesystem directory.

Ajouté dans la version 3.7.

Modifié dans la version 3.12 : Ajout de la gestion de Windows.

`Path.is_socket()`

Return `True` if the path points to a Unix socket. `False` will be returned if the path is invalid, inaccessible or missing, or if it points to something other than a Unix socket. Use `Path.stat()` to distinguish between these cases.

`Path.is_fifo()`

Return `True` if the path points to a FIFO. `False` will be returned if the path is invalid, inaccessible or missing, or if it points to something other than a FIFO. Use `Path.stat()` to distinguish between these cases.

`Path.is_block_device()`

Return `True` if the path points to a block device. `False` will be returned if the path is invalid, inaccessible

or missing, or if it points to something other than a block device. Use `Path.stat()` to distinguish between these cases.

`Path.is_char_device()`

Return `True` if the path points to a character device. `False` will be returned if the path is invalid, inaccessible or missing, or if it points to something other than a character device. Use `Path.stat()` to distinguish between these cases.

`Path.samefile(other_path)`

Renvoie si ce chemin pointe vers le même fichier que *other_path*, qui peut être soit un chemin, soit une chaîne de caractères. La sémantique est similaire à `os.path.samefile()` et `os.path.samestat()`.

`OSError` peut être levée si l'un des fichiers ne peut être accédé pour quelque raison.

```
>>> p = Path('spam')
>>> q = Path('eggs')
>>> p.samefile(q)
False
>>> p.samefile('spam')
True
```

Ajouté dans la version 3.5.

`Path.info`

A `PathInfo` object that supports querying file type information. The object exposes methods that cache their results, which can help reduce the number of system calls needed when switching on file type. For example :

```
>>> p = Path('src')
>>> if p.info.is_symlink():
...     print('symlink')
... elif p.info.is_dir():
...     print('directory')
... elif p.info.exists():
...     print('something else')
... else:
...     print('not found')
...
directory
```

If the path was generated from `Path.iterdir()` then this attribute is initialized with some information about the file type gleaned from scanning the parent directory. Merely accessing `Path.info` does not perform any filesystem queries.

To fetch up-to-date information, it's best to call `Path.is_dir()`, `is_file()` and `is_symlink()` rather than methods of this attribute. There is no way to reset the cache ; instead you can create a new path object with an empty info cache via `p = Path(p)`.

Ajouté dans la version 3.14.

Lecture et écriture de fichiers

`Path.open(mode='r', buffering=-1, encoding=None, errors=None, newline=None)`

Ouvre le fichier pointé par le chemin, comme la fonction native `open()` le fait :

```
>>> p = Path('setup.py')
>>> with p.open() as f:
...     f.readline()
...
'#!/usr/bin/env python3\n'
```

`Path.read_text(encoding=None, errors=None, newline=None)`

Renvoie le contenu décodé du fichier pointé en tant que chaîne de caractères :

```
>>> p = Path('my_text_file')
>>> p.write_text('Text file contents')
18
>>> p.read_text()
'Text file contents'
```

Le fichier est ouvert, puis fermé. Les paramètres optionnels ont la même signification que dans `open()`.

Ajouté dans la version 3.5.

Modifié dans la version 3.13 : ajout du paramètre *newline*.

`Path.read_bytes()`

Renvoie le contenu binaire du fichier pointé en tant que chaîne d'octets :

```
>>> p = Path('my_binary_file')
>>> p.write_bytes(b'Binary file contents')
20
>>> p.read_bytes()
b'Binary file contents'
```

Ajouté dans la version 3.5.

`Path.write_text(data, encoding=None, errors=None, newline=None)`

Ouvre le fichier pointé en mode texte, écrit *data* dedans, et ferme le fichier :

```
>>> p = Path('my_text_file')
>>> p.write_text('Text file contents')
18
>>> p.read_text()
'Text file contents'
```

Le fichier du même nom, s'il existe, est écrasé. Les paramètres optionnels ont la même signification que dans `open()`.

Ajouté dans la version 3.5.

Modifié dans la version 3.10 : ajout du paramètre *newline*.

`Path.write_bytes(data)`

Ouvre le fichier pointé en mode binaire, écrit *data* dedans, et ferme le fichier :

```
>>> p = Path('my_binary_file')
>>> p.write_bytes(b'Binary file contents')
20
>>> p.read_bytes()
b'Binary file contents'
```

Le fichier du même nom, s'il existe, est écrasé.

Ajouté dans la version 3.5.

Lecture des dossiers

`Path.iterdir()`

Quand le chemin pointe vers un dossier, donne par séquences les objets chemin du contenu du dossier :

```
>>> p = Path('docs')
>>> for child in p.iterdir(): child
```

(suite sur la page suivante)

(suite de la page précédente)

```
...
PosixPath('docs/conf.py')
PosixPath('docs/_templates')
PosixPath('docs/make.bat')
PosixPath('docs/index.rst')
PosixPath('docs/_build')
PosixPath('docs/_static')
PosixPath('docs/Makefile')
```

The children are yielded in arbitrary order, and the special entries `'.'` and `'..'` are not included. If a file is removed from or added to the directory after creating the iterator, it is unspecified whether a path object for that file is included.

Si le chemin n'est pas un dossier ou n'est pas accessible, `OSError` est levée.

`Path.glob`(*pattern*, *, *case_sensitive=None*, *recurse_symlinks=False*)

Glob de manière relative suivant le motif *pattern* fourni dans le dossier représenté par ce chemin, donnant tous les fichiers correspondants (de n'importe quelle sorte) :

```
>>> sorted(Path('.').glob('*.py'))
[PosixPath('pathlib.py'), PosixPath('setup.py'), PosixPath('test_pathlib.py')]
>>> sorted(Path('.').glob('*/*.py'))
[PosixPath('docs/conf.py')]
>>> sorted(Path('.').glob('**/*.py'))
[PosixPath('build/lib/pathlib.py'),
 PosixPath('docs/conf.py'),
 PosixPath('pathlib.py'),
 PosixPath('setup.py'),
 PosixPath('test_pathlib.py')]
```

Voir aussi

documentation *Pattern language*.

By default, or when the *case_sensitive* keyword-only argument is set to `None`, this method matches paths using platform-specific casing rules : typically, case-sensitive on POSIX, and case-insensitive on Windows. Set *case_sensitive* to `True` or `False` to override this behaviour.

By default, or when the *recurse_symlinks* keyword-only argument is set to `False`, this method follows symlinks except when expanding "`**`" wildcards. Set *recurse_symlinks* to `True` to always follow symlinks.

Lève un *évènement d'audit* `pathlib.Path.glob` avec comme arguments `self` et `pattern`.

Modifié dans la version 3.12 : ajout du paramètre *case_sensitive*

Modifié dans la version 3.13 : Ajout du paramètre *recurse_symlinks*.

Modifié dans la version 3.13 : Le paramètre *pattern* accepte un *path-like object*.

Modifié dans la version 3.13 : Any `OSError` exceptions raised from scanning the filesystem are suppressed. In previous versions, such exceptions are suppressed in many cases, but not all.

`Path.rglob`(*pattern*, *, *case_sensitive=None*, *recurse_symlinks=False*)

Glob the given relative *pattern* recursively. This is like calling `Path.glob()` with "`**/`" added in front of the *pattern*.

Voir aussi

Documentation pour *Pattern language* et `Path.glob()` .

Lève un *évènement d'audit* `pathlib.Path.glob` avec comme arguments `self` et `pattern`.

Modifié dans la version 3.12 : ajout du paramètre *case_sensitive*

Modifié dans la version 3.13 : Ajout du paramètre *recurse_symlinks*.

Modifié dans la version 3.13 : Le paramètre *pattern* accepte un *path-like object*.

`Path.walk`(*top_down=True*, *on_error=None*, *follow_symlinks=False*)

Generate the file names in a directory tree by walking the tree either top-down or bottom-up.

For each directory in the directory tree rooted at *self* (including *self* but excluding '.' and '..'), the method yields a 3-tuple of `(dirpath, dirnames, filenames)`.

dirpath is a `Path` to the directory currently being walked, *dirnames* is a list of strings for the names of subdirectories in *dirpath* (excluding `'.'` and `'..'`), and *filenames* is a list of strings for the names of the non-directory files in *dirpath*. To get a full path (which begins with *self*) to a file or directory in *dirpath*, do `dirpath / name`. Whether or not the lists are sorted is file system-dependent.

If the optional argument *top_down* is true (which is the default), the triple for a directory is generated before the triples for any of its subdirectories (directories are walked top-down). If *top_down* is false, the triple for a directory is generated after the triples for all of its subdirectories (directories are walked bottom-up). No matter the value of *top_down*, the list of subdirectories is retrieved before the triples for the directory and its subdirectories are walked.

When *top_down* is true, the caller can modify the *dirnames* list in-place (for example, using `del` or slice assignment), and `Path.walk()` will only recurse into the subdirectories whose names remain in *dirnames*. This can be used to prune the search, or to impose a specific order of visiting, or even to inform `Path.walk()` about directories the caller creates or renames before it resumes `Path.walk()` again. Modifying *dirnames* when *top_down* is false has no effect on the behavior of `Path.walk()` since the directories in *dirnames* have already been generated by the time *dirnames* is yielded to the caller.

By default, errors from `os.scandir()` are ignored. If the optional argument *on_error* is specified, it should be a callable ; it will be called with one argument, an `OSError` instance. The callable can handle the error to continue the walk or re-raise it to stop the walk. Note that the filename is available as the `filename` attribute of the exception object.

By default, `Path.walk()` does not follow symbolic links, and instead adds them to the *filenames* list. Set *follow_symlinks* to true to resolve symlinks and place them in *dirnames* and *filenames* as appropriate for their targets, and consequently visit directories pointed to by symlinks (where supported).

> **Note**
>
> Be aware that setting *follow_symlinks* to true can lead to infinite recursion if a link points to a parent directory of itself. `Path.walk()` does not keep track of the directories it has already visited.

> **Note**
>
> `Path.walk()` assumes the directories it walks are not modified during execution. For example, if a directory from *dirnames* has been replaced with a symlink and *follow_symlinks* is false, `Path.walk()` will still try to descend into it. To prevent such behavior, remove directories from *dirnames* as appropriate.

> **Note**
>
> Unlike `os.walk()`, `Path.walk()` lists symlinks to directories in *filenames* if *follow_symlinks* is false.

This example displays the number of bytes used by all files in each directory, while ignoring `__pycache__` directories :

```
from pathlib import Path
for root, dirs, files in Path("cpython/Lib/concurrent").walk(on_error=print):
  print(
      root,
      "consumes",
      sum((root / file).stat().st_size for file in files),
      "bytes in",
      len(files),
      "non-directory files"
  )
  if '__pycache__' in dirs:
        dirs.remove('__pycache__')
```

This next example is a simple implementation of `shutil.rmtree()`. Walking the tree bottom-up is essential as `rmdir()` doesn't allow deleting a directory before it is empty :

```
# Delete everything reachable from the directory "top".
# CAUTION:  This is dangerous! For example, if top == Path('/'),
# it could delete all of your files.
for root, dirs, files in top.walk(top_down=False):
    for name in files:
        (root / name).unlink()
    for name in dirs:
        (root / name).rmdir()
```

Ajouté dans la version 3.12.

Création de fichiers et de dossiers

`Path.touch`(*mode=0o666*, *exist_ok=True*)

Create a file at this given path. If *mode* is given, it is combined with the process's `umask` value to determine the file mode and access flags. If the file already exists, the function succeeds when *exist_ok* is true (and its modification time is updated to the current time), otherwise `FileExistsError` is raised.

> **Voir aussi**
>
> The `open()`, `write_text()` and `write_bytes()` methods are often used to create files.

`Path.mkdir`(*mode=0o777*, *parents=False*, *exist_ok=False*)

Create a new directory at this given path. If *mode* is given, it is combined with the process's `umask` value to determine the file mode and access flags. If the path already exists, `FileExistsError` is raised.

Si *parents* est vrai, chaque parent de ce chemin est créé si besoin ; ils sont créés avec les permissions par défaut sans prendre en compte *mode* (reproduisant la commande POSIX `mkdir -p`).

Si *parents* est faux (valeur par défaut), un parent manquant lève `FileNotFoundError`.

Si *exist_ok* est faux (valeur par défaut), `FileExistsError` est levé si le dossier cible existe déjà.

If *exist_ok* is true, `FileExistsError` will not be raised unless the given path already exists in the file system and is not a directory (same behavior as the POSIX `mkdir -p` command).

Modifié dans la version 3.5 : ajout du paramètre *exist_ok*.

`Path.symlink_to`(*target*, *target_is_directory=False*)

Make this path a symbolic link pointing to *target*.

On Windows, a symlink represents either a file or a directory, and does not morph to the target dynamically. If the target is present, the type of the symlink will be created to match. Otherwise, the symlink will be created as a directory if *target_is_directory* is true or a file symlink (the default) otherwise. On non-Windows platforms,

target_is_directory is ignored.

```
>>> p = Path('mylink')
>>> p.symlink_to('setup.py')
>>> p.resolve()
PosixPath('/home/antoine/pathlib/setup.py')
>>> p.stat().st_size
956
>>> p.lstat().st_size
8
```

> **Note**
>
> L'ordre des arguments (lien, cible) est l'opposé de ceux de `os.symlink()`.

Modifié dans la version 3.13 : Raises `UnsupportedOperation` if `os.symlink()` is not available. In previous versions, `NotImplementedError` was raised.

`Path.`**`hardlink_to`**(*target*)

Crée un lien physique pointant sur le même fichier que *target*.

> **Note**
>
> L'ordre des arguments (lien, cible) est l'opposé de celui de `os.link()`.

Ajouté dans la version 3.10.

Modifié dans la version 3.13 : Raises `UnsupportedOperation` if `os.link()` is not available. In previous versions, `NotImplementedError` was raised.

Copying, moving and deleting

`Path.`**`copy`**(*target*, *, *follow_symlinks=True*, *dirs_exist_ok=False*, *preserve_metadata=False*)

Copy this file or directory tree to the given *target*, and return a new `Path` instance pointing to *target*.

If the source is a file, the target will be replaced if it is an existing file. If the source is a symlink and *follow_symlinks* is true (the default), the symlink's target is copied. Otherwise, the symlink is recreated at the destination.

If the source is a directory and *dirs_exist_ok* is false (the default), a `FileExistsError` is raised if the target is an existing directory. If *dirs_exists_ok* is true, the copying operation will overwrite existing files within the destination tree with corresponding files from the source tree.

If *preserve_metadata* is false (the default), only directory structures and file data are guaranteed to be copied. Set *preserve_metadata* to true to ensure that file and directory permissions, flags, last access and modification times, and extended attributes are copied where supported. This argument has no effect when copying files on Windows (where metadata is always preserved).

> **Note**
>
> Where supported by the operating system and file system, this method performs a lightweight copy, where data blocks are only copied when modified. This is known as copy-on-write.

Ajouté dans la version 3.14.

`Path.`**`copy_into`**(*target_dir*, *, *follow_symlinks=True*, *dirs_exist_ok=False*, *preserve_metadata=False*)

Copy this file or directory tree into the given *target_dir*, which should be an existing directory. Other arguments are handled identically to `Path.copy()`. Returns a new `Path` instance pointing to the copy.

Ajouté dans la version 3.14.

`Path.rename(target)`

Rename this file or directory to the given *target*, and return a new `Path` instance pointing to *target*. On Unix, if *target* exists and is a file, it will be replaced silently if the user has permission. On Windows, if *target* exists, `FileExistsError` will be raised. *target* can be either a string or another path object :

```
>>> p = Path('foo')
>>> p.open('w').write('some text')
9
>>> target = Path('bar')
>>> p.rename(target)
PosixPath('bar')
>>> target.open().read()
'some text'
```

The target path may be absolute or relative. Relative paths are interpreted relative to the current working directory, *not* the directory of the `Path` object.

Cette méthode est implémentée d'après `os.rename()` et fournit les mêmes garanties.

Modifié dans la version 3.8 : Added return value, return the new `Path` instance.

`Path.replace(target)`

Rename this file or directory to the given *target*, and return a new `Path` instance pointing to *target*. If *target* points to an existing file or empty directory, it will be unconditionally replaced.

The target path may be absolute or relative. Relative paths are interpreted relative to the current working directory, *not* the directory of the `Path` object.

Modifié dans la version 3.8 : Added return value, return the new `Path` instance.

`Path.move(target)`

Move this file or directory tree to the given *target*, and return a new `Path` instance pointing to *target*.

If the *target* doesn't exist it will be created. If both this path and the *target* are existing files, then the target is overwritten. If both paths point to the same file or directory, or the *target* is a non-empty directory, then `OSError` is raised.

If both paths are on the same filesystem, the move is performed with `os.replace()`. Otherwise, this path is copied (preserving metadata and symlinks) and then deleted.

Ajouté dans la version 3.14.

`Path.move_into(target_dir)`

Move this file or directory tree into the given *target_dir*, which should be an existing directory. Returns a new `Path` instance pointing to the moved path.

Ajouté dans la version 3.14.

`Path.unlink(missing_ok=False)`

Supprime ce fichier ou lien symbolique. Si le chemin pointe vers un dossier, utilisez `Path.rmdir()` à la place.

Si *missing_ok* est faux (valeur par défaut), une `FileNotFoundError` est levée si le chemin n'existe pas.

Si *missing_ok* est vrai, les exceptions `FileNotFoundError` sont ignorées (même comportement que la commande POSIX `rm -f`).

Modifié dans la version 3.8 : Ajout du paramètre *missing_ok*.

`Path.rmdir()`

Supprime ce dossier. Le dossier doit être vide.

Permissions et propriété

`Path.owner`(*, *follow_symlinks=True*)

Return the name of the user owning the file. `KeyError` is raised if the file's user identifier (UID) isn't found in the system database.

This method normally follows symlinks ; to get the owner of the symlink, add the argument `follow_symlinks=False`.

Modifié dans la version 3.13 : Raises `UnsupportedOperation` if the `pwd` module is not available. In earlier versions, `NotImplementedError` was raised.

Modifié dans la version 3.13 : ajout du paramètre *follow_symlinks*.

`Path.group`(*, *follow_symlinks=True*)

Return the name of the group owning the file. `KeyError` is raised if the file's group identifier (GID) isn't found in the system database.

This method normally follows symlinks ; to get the group of the symlink, add the argument `follow_symlinks=False`.

Modifié dans la version 3.13 : Raises `UnsupportedOperation` if the `grp` module is not available. In earlier versions, `NotImplementedError` was raised.

Modifié dans la version 3.13 : ajout du paramètre *follow_symlinks*.

`Path.chmod`(*mode*, *, *follow_symlinks=True*)

Change le mode et les permissions du fichier, comme `os.chmod()`.

Cette méthode suit normalement les liens symboliques. Certaines versions d'Unix prennent en charge la modification des permissions sur le lien symbolique lui-même ; sur ces plateformes, vous pouvez ajouter l'argument `follow_symlinks=False`, ou utiliser `lchmod()`.

```
>>> p = Path('setup.py')
>>> p.stat().st_mode
33277
>>> p.chmod(0o444)
>>> p.stat().st_mode
33060
```

Modifié dans la version 3.10 : ajout du paramètre *follow_symlinks*.

`Path.lchmod`(*mode*)

Comme `Path.chmod()`, mais, si le chemin pointe vers un lien symbolique, le mode du lien symbolique est changé plutôt que celui de sa cible.

11.1.5 Pattern language

The following wildcards are supported in patterns for `full_match()`, `glob()` and `rglob()` :

`**` (tout le segment)
Matche un nombre quelconque de segments de fichiers ou dossiers, zéro inclus

`*` (tout le segment)
Correspondance d'un segment de fichier ou de dossier

`*` (partie du segment)
Matches any number of non-separator characters, including zero.

`?`
Matches one non-separator character.

`[seq]`
Correspondance d'un caractère dans *seq*

`[!seq]`
Matches one character not in *seq*.

For a literal match, wrap the meta-characters in brackets. For example, "[?]" matches the character "?".

The "**" wildcard enables recursive globbing. A few examples :

Motif	Signification
"**/*"	N'importe quel chemin avec au moins un segment
"**/*.py"	Any path with a final segment ending ".py".
"assets/**"	Any path starting with "assets/".
"assets/**/*"	Any path starting with "assets/", excluding "assets/" itself.

> **Note**
>
> Globbing with the "**" wildcard visits every directory in the tree. Large directory trees may take a long time to search.

Modifié dans la version 3.13 : Globbing with a pattern that ends with "**" returns both files and directories. In previous versions, only directories were returned.

In `Path.glob()` and `rglob()`, a trailing slash may be added to the pattern to match only directories.

Modifié dans la version 3.11 : Globbing with a pattern that ends with a pathname components separator (`sep` or `altsep`) returns only directories.

11.1.6 Comparaison avec les outils du module `glob`

The patterns accepted and results generated by `Path.glob()` and `Path.rglob()` differ slightly from those by the `glob` module :

1. Files beginning with a dot are not special in pathlib. This is like passing `include_hidden=True` to `glob.glob()`.
2. "**" pattern components are always recursive in pathlib. This is like passing `recursive=True` to `glob.glob()`.
3. "**" pattern components do not follow symlinks by default in pathlib. This behaviour has no equivalent in `glob.glob()`, but you can pass `recurse_symlinks=True` to `Path.glob()` for compatible behaviour.
4. Like all `PurePath` and `Path` objects, the values returned from `Path.glob()` and `Path.rglob()` don't include trailing slashes.
5. The values returned from pathlib's `path.glob()` and `path.rglob()` include the *path* as a prefix, unlike the results of `glob.glob(root_dir=path)`.
6. The values returned from pathlib's `path.glob()` and `path.rglob()` may include *path* itself, for example when globbing "**", whereas the results of `glob.glob(root_dir=path)` never include an empty string that would correspond to *path*.

11.1.7 Comparaison aux modules `os` et `os.path`

pathlib implements path operations using `PurePath` and `Path` objects, and so it's said to be *object-oriented*. On the other hand, the `os` and `os.path` modules supply functions that work with low-level `str` and `bytes` objects, which is a more *procedural* approach. Some users consider the object-oriented style to be more readable.

Many functions in `os` and `os.path` support `bytes` paths and *paths relative to directory descriptors*. These features aren't available in pathlib.

Python's `str` and `bytes` types, and portions of the `os` and `os.path` modules, are written in C and are very speedy. pathlib is written in pure Python and is often slower, but rarely slow enough to matter.

pathlib's path normalization is slightly more opinionated and consistent than `os.path`. For example, whereas `os.path.abspath()` eliminates ".." segments from a path, which may change its meaning if symlinks are involved, `Path.absolute()` preserves these segments for greater safety.

pathlib's path normalization may render it unsuitable for some applications :

1. pathlib normalizes `Path("my_folder/")` to `Path("my_folder")`, which changes a path's meaning when supplied to various operating system APIs and command-line utilities. Specifically, the absence of a trailing separator may allow the path to be resolved as either a file or directory, rather than a directory only.
2. pathlib normalizes `Path("./my_program")` to `Path("my_program")`, which changes a path's meaning when used as an executable search path, such as in a shell or when spawning a child process. Specifically, the absence of a separator in the path may force it to be looked up in `PATH` rather than the current directory.

As a consequence of these differences, pathlib is not a drop-in replacement for `os.path`.

Outils correspondants

Ci-dessous se trouve un tableau associant diverses fonctions `os` à leur équivalent `PurePath` / `Path` correspondant.

`os` et `os.path`	`pathlib`
`os.path.dirname()`	`PurePath.parent`
`os.path.basename()`	`PurePath.name`
`os.path.splitext()`	`PurePath.stem`, `PurePath.suffix`
`os.path.join()`	`PurePath.joinpath()`
`os.path.isabs()`	`PurePath.is_absolute()`
`os.path.relpath()`	`PurePath.relative_to()` [1]
`os.path.expanduser()`	`Path.expanduser()` [2]
`os.path.realpath()`	`Path.resolve()`
`os.path.abspath()`	`Path.absolute()` [3]
`os.path.exists()`	`Path.exists()`
`os.path.isfile()`	`Path.is_file()`
`os.path.isdir()`	`Path.is_dir()`
`os.path.islink()`	`Path.is_symlink()`
`os.path.isjunction()`	`Path.is_junction()`
`os.path.ismount()`	`Path.is_mount()`
`os.path.samefile()`	`Path.samefile()`
`os.getcwd()`	`Path.cwd()`
`os.stat()`	`Path.stat()`
`os.lstat()`	`Path.lstat()`
`os.listdir()`	`Path.iterdir()`
`os.walk()`	`Path.walk()` [4]
`os.mkdir()`, `os.makedirs()`	`Path.mkdir()`
`os.link()`	`Path.hardlink_to()`
`os.symlink()`	`Path.symlink_to()`
`os.readlink()`	`Path.readlink()`
`os.rename()`	`Path.rename()`
`os.replace()`	`Path.replace()`
`os.remove()`, `os.unlink()`	`Path.unlink()`
`os.rmdir()`	`Path.rmdir()`
`os.chmod()`	`Path.chmod()`
`os.lchmod()`	`Path.lchmod()`

1. `os.path.relpath()` calls `abspath()` to make paths absolute and remove ".." parts, whereas `PurePath.relative_to()` is a lexical operation that raises `ValueError` when its inputs' anchors differ (e.g. if one path is absolute and the other relative.)
2. `os.path.expanduser()` returns the path unchanged if the home directory can't be resolved, whereas `Path.expanduser()` raises `RuntimeError`.
3. `os.path.abspath()` removes ".." components without resolving symlinks, which may change the meaning of the path, whereas `Path.absolute()` leaves any ".." components in the path.
4. `os.walk()` always follows symlinks when categorizing paths into *dirnames* and *filenames*, whereas `Path.walk()` categorizes all symlinks into *filenames* when *follow_symlinks* is false (the default.)

Notes

11.1.8 Protocols

The `pathlib.types` module provides types for static type checking.

Ajouté dans la version 3.14.

class pathlib.types.**PathInfo**

A `typing.Protocol` describing the `Path.info` attribute. Implementations may return cached results from their methods.

exists(*, *follow_symlinks=True*)

Return `True` if the path is an existing file or directory, or any other kind of file ; return `False` if the path doesn't exist.

If *follow_symlinks* is `False`, return `True` for symlinks without checking if their targets exist.

is_dir(*, *follow_symlinks=True*)

Return `True` if the path is a directory, or a symbolic link pointing to a directory ; return `False` if the path is (or points to) any other kind of file, or if it doesn't exist.

If *follow_symlinks* is `False`, return `True` only if the path is a directory (without following symlinks) ; return `False` if the path is any other kind of file, or if it doesn't exist.

is_file(*, *follow_symlinks=True*)

Return `True` if the path is a file, or a symbolic link pointing to a file ; return `False` if the path is (or points to) a directory or other non-file, or if it doesn't exist.

If *follow_symlinks* is `False`, return `True` only if the path is a file (without following symlinks) ; return `False` if the path is a directory or other other non-file, or if it doesn't exist.

is_symlink()

Return `True` if the path is a symbolic link (even if broken) ; return `False` if the path is a directory or any kind of file, or if it doesn't exist.

11.2 `os.path` --- Common pathname manipulations

Source code : Lib/genericpath.py, Lib/posixpath.py (for POSIX) and Lib/ntpath.py (for Windows).

This module implements some useful functions on pathnames. To read or write files see `open()`, and for accessing the filesystem see the `os` module. The path parameters can be passed as strings, or bytes, or any object implementing the `os.PathLike` protocol.

Unlike a Unix shell, Python does not do any *automatic* path expansions. Functions such as `expanduser()` and `expandvars()` can be invoked explicitly when an application desires shell-like path expansion. (See also the `glob` module.)

Voir aussi

Le module `pathlib` offre une représentation objet de haut niveau des chemins.

Note

Toutes ces fonctions n'acceptent que des chaînes d'octets ou des chaînes de caractères en tant que paramètres. Le résultat est un objet du même type si un chemin ou un nom de fichier est renvoyé.

> **Note**
>
> Comme les différents systèmes d'exploitation ont des conventions de noms de chemins différentes, il existe plusieurs versions de ce module dans la bibliothèque standard. Le module `os.path` est toujours le module de chemin adapté au système d'exploitation sur lequel Python tourne, et donc adapté pour les chemins locaux. Cependant, vous pouvez également importer et utiliser les modules individuels si vous voulez manipuler un chemin qui est *toujours* dans l'un des différents formats. Ils ont tous la même interface :
>
> — `posixpath` pour les chemins de type UNIX
> — `ntpath` pour les chemins Windows

Modifié dans la version 3.8 : `exists()`, `lexists()`, `isdir()`, `isfile()`, `islink()`, and `ismount()` now return `False` instead of raising an exception for paths that contain characters or bytes unrepresentable at the OS level.

`os.path.`**`abspath`**(*path*)

Renvoie une version absolue et normalisée du chemin d'accès *path*. Sur la plupart des plates-formes, cela équivaut à appeler la fonction `normpath()` comme suit : `normpath(join(os.getcwd(), chemin))`.

Modifié dans la version 3.6 : Accepte un *path-like object*.

`os.path.`**`basename`**(*path*)

Renvoie le nom de base du chemin d'accès *path*. C'est le second élément de la paire renvoyée en passant *path* à la fonction `split()`. Notez que le résultat de cette fonction est différent de celui du programme Unix **`basename`** ; là où **`basename`** pour `'/foo/bar/'` renvoie `'bar'`, la fonction `basename()` renvoie une chaîne vide (`''`).

Modifié dans la version 3.6 : Accepte un *path-like object*.

`os.path.`**`commonpath`**(*paths*)

Return the longest common sub-path of each pathname in the iterable *paths*. Raise `ValueError` if *paths* contain both absolute and relative pathnames, if *paths* are on different drives, or if *paths* is empty. Unlike `commonprefix()`, this returns a valid path.

Ajouté dans la version 3.5.

Modifié dans la version 3.6 : Accepts a sequence of *path-like objects*.

Modifié dans la version 3.13 : Any iterable can now be passed, rather than just sequences.

`os.path.`**`commonprefix`**(*list*)

Return the longest path prefix (taken character-by-character) that is a prefix of all paths in *list*. If *list* is empty, return the empty string (`''`).

> **Note**
>
> This function may return invalid paths because it works a character at a time. To obtain a valid path, see `commonpath()`.
>
> ```
> >>> os.path.commonprefix(['/usr/lib', '/usr/local/lib'])
> '/usr/l'
>
> >>> os.path.commonpath(['/usr/lib', '/usr/local/lib'])
> '/usr'
> ```

Modifié dans la version 3.6 : Accepte un *path-like object*.

`os.path.`**`dirname`**(*path*)

Return the directory name of pathname *path*. This is the first element of the pair returned by passing *path* to the function `split()`.

Modifié dans la version 3.6 : Accepte un *path-like object*.

os.path.**exists**(*path*)

Return True if *path* refers to an existing path or an open file descriptor. Returns False for broken symbolic links. On some platforms, this function may return False if permission is not granted to execute os.stat() on the requested file, even if the *path* physically exists.

Modifié dans la version 3.3 : *path* can now be an integer : True is returned if it is an open file descriptor, False otherwise.

Modifié dans la version 3.6 : Accepte un *path-like object*.

os.path.**lexists**(*path*)

Return True if *path* refers to an existing path, including broken symbolic links. Equivalent to exists() on platforms lacking os.lstat().

Modifié dans la version 3.6 : Accepte un *path-like object*.

os.path.**expanduser**(*path*)

On Unix and Windows, return the argument with an initial component of ~ or ~user replaced by that *user*'s home directory.

On Unix, an initial ~ is replaced by the environment variable HOME if it is set; otherwise the current user's home directory is looked up in the password directory through the built-in module pwd. An initial ~user is looked up directly in the password directory.

On Windows, USERPROFILE will be used if set, otherwise a combination of HOMEPATH and HOMEDRIVE will be used. An initial ~user is handled by checking that the last directory component of the current user's home directory matches USERNAME, and replacing it if so.

If the expansion fails or if the path does not begin with a tilde, the path is returned unchanged.

Modifié dans la version 3.6 : Accepte un *path-like object*.

Modifié dans la version 3.8 : No longer uses HOME on Windows.

os.path.**expandvars**(*path*)

Return the argument with environment variables expanded. Substrings of the form $name or ${name} are replaced by the value of environment variable *name*. Malformed variable names and references to non-existing variables are left unchanged.

On Windows, %name% expansions are supported in addition to $name and ${name}.

Modifié dans la version 3.6 : Accepte un *path-like object*.

os.path.**getatime**(*path*)

Return the time of last access of *path*. The return value is a floating-point number giving the number of seconds since the epoch (see the time module). Raise OSError if the file does not exist or is inaccessible.

os.path.**getmtime**(*path*)

Return the time of last modification of *path*. The return value is a floating-point number giving the number of seconds since the epoch (see the time module). Raise OSError if the file does not exist or is inaccessible.

Modifié dans la version 3.6 : Accepte un *path-like object*.

os.path.**getctime**(*path*)

Return the system's ctime which, on some systems (like Unix) is the time of the last metadata change, and, on others (like Windows), is the creation time for *path*. The return value is a number giving the number of seconds since the epoch (see the time module). Raise OSError if the file does not exist or is inaccessible.

Modifié dans la version 3.6 : Accepte un *path-like object*.

os.path.**getsize**(*path*)

Return the size, in bytes, of *path*. Raise OSError if the file does not exist or is inaccessible.

Modifié dans la version 3.6 : Accepte un *path-like object*.

os.path.**isabs**(*path*)

Return `True` if *path* is an absolute pathname. On Unix, that means it begins with a slash, on Windows that it begins with two (back)slashes, or a drive letter, colon, and (back)slash together.

Modifié dans la version 3.6 : Accepte un *path-like object*.

Modifié dans la version 3.13 : On Windows, returns `False` if the given path starts with exactly one (back)slash.

os.path.**isfile**(*path*)

Return `True` if *path* is an *existing* regular file. This follows symbolic links, so both *islink()* and *isfile()* can be true for the same path.

Modifié dans la version 3.6 : Accepte un *path-like object*.

os.path.**isdir**(*path*)

Return `True` if *path* is an *existing* directory. This follows symbolic links, so both *islink()* and *isdir()* can be true for the same path.

Modifié dans la version 3.6 : Accepte un *path-like object*.

os.path.**isjunction**(*path*)

Return `True` if *path* refers to an *existing* directory entry that is a junction. Always return `False` if junctions are not supported on the current platform.

Ajouté dans la version 3.12.

os.path.**islink**(*path*)

Return `True` if *path* refers to an *existing* directory entry that is a symbolic link. Always `False` if symbolic links are not supported by the Python runtime.

Modifié dans la version 3.6 : Accepte un *path-like object*.

os.path.**ismount**(*path*)

Return `True` if pathname *path* is a *mount point* : a point in a file system where a different file system has been mounted. On POSIX, the function checks whether *path*'s parent, `path/..`, is on a different device than *path*, or whether `path/..` and *path* point to the same i-node on the same device --- this should detect mount points for all Unix and POSIX variants. It is not able to reliably detect bind mounts on the same filesystem. On Windows, a drive letter root and a share UNC are always mount points, and for any other path `GetVolumePathName` is called to see if it is different from the input path.

Modifié dans la version 3.4 : Added support for detecting non-root mount points on Windows.

Modifié dans la version 3.6 : Accepte un *path-like object*.

os.path.**isdevdrive**(*path*)

Return `True` if pathname *path* is located on a Windows Dev Drive. A Dev Drive is optimized for developer scenarios, and offers faster performance for reading and writing files. It is recommended for use for source code, temporary build directories, package caches, and other IO-intensive operations.

May raise an error for an invalid path, for example, one without a recognizable drive, but returns `False` on platforms that do not support Dev Drives. See the Windows documentation for information on enabling and creating Dev Drives.

Ajouté dans la version 3.12.

Modifié dans la version 3.13 : The function is now available on all platforms, and will always return `False` on those that have no support for Dev Drives

os.path.**isreserved**(*path*)

Return `True` if *path* is a reserved pathname on the current system.

On Windows, reserved filenames include those that end with a space or dot ; those that contain colons (i.e. file streams such as "name :stream"), wildcard characters (i.e. `'*?"<>'`), pipe, or ASCII control characters ; as well as DOS device names such as "NUL", "CON", "CONIN$", "CONOUT$", "AUX", "PRN", "COM1", and "LPT1".

> **Note**
>
> This function approximates rules for reserved paths on most Windows systems. These rules change over time in various Windows releases. This function may be updated in future Python releases as changes to the rules become broadly available.

Availability : Windows.

Ajouté dans la version 3.13.

`os.path.join`(*path*, **paths*)

Join one or more path segments intelligently. The return value is the concatenation of *path* and all members of **paths*, with exactly one directory separator following each non-empty part, except the last. That is, the result will only end in a separator if the last part is either empty or ends in a separator. If a segment is an absolute path (which on Windows requires both a drive and a root), then all previous segments are ignored and joining continues from the absolute path segment.

On Windows, the drive is not reset when a rooted path segment (e.g., `r'\foo'`) is encountered. If a segment is on a different drive or is an absolute path, all previous segments are ignored and the drive is reset. Note that since there is a current directory for each drive, `os.path.join("c:", "foo")` represents a path relative to the current directory on drive `C:` (`c:foo`), not `c:\foo`.

Modifié dans la version 3.6 : Accepts a *path-like object* for *path* and *paths*.

`os.path.normcase`(*path*)

Normalize the case of a pathname. On Windows, convert all characters in the pathname to lowercase, and also convert forward slashes to backward slashes. On other operating systems, return the path unchanged.

Modifié dans la version 3.6 : Accepte un *path-like object*.

`os.path.normpath`(*path*)

Normalize a pathname by collapsing redundant separators and up-level references so that `A//B`, `A/B/`, `A/./B` and `A/foo/../B` all become `A/B`. This string manipulation may change the meaning of a path that contains symbolic links. On Windows, it converts forward slashes to backward slashes. To normalize case, use `normcase()`.

> **Note**
>
> On POSIX systems, in accordance with IEEE Std 1003.1 2013 Edition; 4.13 Pathname Resolution, if a pathname begins with exactly two slashes, the first component following the leading characters may be interpreted in an implementation-defined manner, although more than two leading characters shall be treated as a single character.

Modifié dans la version 3.6 : Accepte un *path-like object*.

`os.path.realpath`(*path*, ***, *strict=False*)

Return the canonical path of the specified filename, eliminating any symbolic links encountered in the path (if they are supported by the operating system). On Windows, this function will also resolve MS-DOS (also called 8.3) style names such as `C:\\PROGRA~1` to `C:\\Program Files`.

If a path doesn't exist or a symlink loop is encountered, and *strict* is `True`, `OSError` is raised. If *strict* is `False` these errors are ignored, and so the result might be missing or otherwise inaccessible.

> **Note**
>
> This function emulates the operating system's procedure for making a path canonical, which differs slightly between Windows and UNIX with respect to how links and subsequent path components interact.
>
> Operating system APIs make paths canonical as needed, so it's not normally necessary to call this function.

Modifié dans la version 3.6 : Accepte un *path-like object*.

Modifié dans la version 3.8 : Symbolic links and junctions are now resolved on Windows.

Modifié dans la version 3.10 : The *strict* parameter was added.

`os.path.`**`relpath`**(*path*, *start=os.curdir*)

Return a relative filepath to *path* either from the current directory or from an optional *start* directory. This is a path computation : the filesystem is not accessed to confirm the existence or nature of *path* or *start*. On Windows, `ValueError` is raised when *path* and *start* are on different drives.

start defaults to `os.curdir`.

Modifié dans la version 3.6 : Accepte un *path-like object*.

`os.path.`**`samefile`**(*path1*, *path2*)

Return `True` if both pathname arguments refer to the same file or directory. This is determined by the device number and i-node number and raises an exception if an `os.stat()` call on either pathname fails.

Modifié dans la version 3.2 : Prise en charge de Windows.

Modifié dans la version 3.4 : Windows now uses the same implementation as all other platforms.

Modifié dans la version 3.6 : Accepte un *path-like object*.

`os.path.`**`sameopenfile`**(*fp1*, *fp2*)

Return `True` if the file descriptors *fp1* and *fp2* refer to the same file.

Modifié dans la version 3.2 : Prise en charge de Windows.

Modifié dans la version 3.6 : Accepte un *path-like object*.

`os.path.`**`samestat`**(*stat1*, *stat2*)

Return `True` if the stat tuples *stat1* and *stat2* refer to the same file. These structures may have been returned by `os.fstat()`, `os.lstat()`, or `os.stat()`. This function implements the underlying comparison used by `samefile()` and `sameopenfile()`.

Modifié dans la version 3.4 : Prise en charge de Windows.

Modifié dans la version 3.6 : Accepte un *path-like object*.

`os.path.`**`split`**(*path*)

Split the pathname *path* into a pair, `(head, tail)` where *tail* is the last pathname component and *head* is everything leading up to that. The *tail* part will never contain a slash ; if *path* ends in a slash, *tail* will be empty. If there is no slash in *path*, *head* will be empty. If *path* is empty, both *head* and *tail* are empty. Trailing slashes are stripped from *head* unless it is the root (one or more slashes only). In all cases, `join(head, tail)` returns a path to the same location as *path* (but the strings may differ). Also see the functions `dirname()` and `basename()`.

Modifié dans la version 3.6 : Accepte un *path-like object*.

`os.path.`**`splitdrive`**(*path*)

Split the pathname *path* into a pair `(drive, tail)` where *drive* is either a mount point or the empty string. On systems which do not use drive specifications, *drive* will always be the empty string. In all cases, `drive + tail` will be the same as *path*.

On Windows, splits a pathname into drive/UNC sharepoint and relative path.

If the path contains a drive letter, drive will contain everything up to and including the colon :

```
>>> splitdrive("c:/dir")
("c:", "/dir")
```

If the path contains a UNC path, drive will contain the host name and share :

```
>>> splitdrive("//host/computer/dir")
("//host/computer", "/dir")
```

Modifié dans la version 3.6 : Accepte un *path-like object*.

os.path.**splitroot**(*path*)

Split the pathname *path* into a 3-item tuple `(drive, root, tail)` where *drive* is a device name or mount point, *root* is a string of separators after the drive, and *tail* is everything after the root. Any of these items may be the empty string. In all cases, `drive + root + tail` will be the same as *path*.

On POSIX systems, *drive* is always empty. The *root* may be empty (if *path* is relative), a single forward slash (if *path* is absolute), or two forward slashes (implementation-defined per IEEE Std 1003.1-2017 ; 4.13 Pathname Resolution.) For example :

```
>>> splitroot('/home/sam')
('', '/', 'home/sam')
>>> splitroot('//home/sam')
('', '//', 'home/sam')
>>> splitroot('///home/sam')
('', '/', '//home/sam')
```

On Windows, *drive* may be empty, a drive-letter name, a UNC share, or a device name. The *root* may be empty, a forward slash, or a backward slash. For example :

```
>>> splitroot('C:/Users/Sam')
('C:', '/', 'Users/Sam')
>>> splitroot('//Server/Share/Users/Sam')
('//Server/Share', '/', 'Users/Sam')
```

Ajouté dans la version 3.12.

os.path.**splitext**(*path*)

Split the pathname *path* into a pair `(root, ext)` such that `root + ext == path`, and the extension, *ext*, is empty or begins with a period and contains at most one period.

If the path contains no extension, *ext* will be `''` :

```
>>> splitext('bar')
('bar', '')
```

If the path contains an extension, then *ext* will be set to this extension, including the leading period. Note that previous periods will be ignored :

```
>>> splitext('foo.bar.exe')
('foo.bar', '.exe')
>>> splitext('/foo/bar.exe')
('/foo/bar', '.exe')
```

Leading periods of the last component of the path are considered to be part of the root :

```
>>> splitext('.cshrc')
('.cshrc', '')
>>> splitext('/foo/....jpg')
('/foo/....jpg', '')
```

Modifié dans la version 3.6 : Accepte un *path-like object*.

os.path.**supports_unicode_filenames**

`True` if arbitrary Unicode strings can be used as file names (within limitations imposed by the file system).

11.3 stat --- Interpreting stat() results

Code source : Lib/stat.py

The *stat* module defines constants and functions for interpreting the results of *os.stat()*, *os.fstat()* and *os.lstat()* (if they exist). For complete details about the stat(), fstat() and lstat() calls, consult the documentation for your system.

Modifié dans la version 3.4 : Le module *stat* est implémenté en C.

Le module *stat* définit les fonctions suivantes pour tester des types de fichiers spécifiques :

stat.**S_ISDIR**(*mode*)

Renvoie une valeur différente de zéro si c'est un mode d'un répertoire.

stat.**S_ISCHR**(*mode*)

Renvoie une valeur différente de zéro si c'est le mode d'un fichier spécial caractère de périphérique.

stat.**S_ISBLK**(*mode*)

Renvoie une valeur différente de zéro si c'est le mode d'un fichier spécial bloc.

stat.**S_ISREG**(*mode*)

Renvoie une valeur différente de zéro si c'est le mode d'un fichier normal.

stat.**S_ISFIFO**(*mode*)

Renvoie une valeur différente de zéro si c'est le mode d'une FIFO (tube nommé)

stat.**S_ISLNK**(*mode*)

Renvoie une valeur différente de zéro si c'est le mode d'un lien symbolique.

stat.**S_ISSOCK**(*mode*)

Renvoie une valeur différente de zéro si c'est le mode d'un *socket*.

stat.**S_ISDOOR**(*mode*)

Return non-zero if the mode is from a door.

Ajouté dans la version 3.4.

stat.**S_ISPORT**(*mode*)

Return non-zero if the mode is from an event port.

Ajouté dans la version 3.4.

stat.**S_ISWHT**(*mode*)

Return non-zero if the mode is from a whiteout.

Ajouté dans la version 3.4.

Deux autres fonctions sont définies pour permettre plus de manipulation du mode du fichier :

stat.**S_IMODE**(*mode*)

Return the portion of the file's mode that can be set by *os.chmod()*---that is, the file's permission bits, plus the sticky bit, set-group-id, and set-user-id bits (on systems that support them).

stat.**S_IFMT**(*mode*)

Return the portion of the file's mode that describes the file type (used by the S_IS*() functions above).

Normally, you would use the os.path.is*() functions for testing the type of a file ; the functions here are useful when you are doing multiple tests of the same file and wish to avoid the overhead of the stat() system call for each test. These are also useful when checking for information about a file that isn't handled by *os.path*, like the tests for block and character devices.

Exemple :

```
import os, sys
from stat import *

def walktree(top, callback):
    '''recursively descend the directory tree rooted at top,
       calling the callback function for each regular file'''

    for f in os.listdir(top):
        pathname = os.path.join(top, f)
        mode = os.lstat(pathname).st_mode
        if S_ISDIR(mode):
            # It's a directory, recurse into it
            walktree(pathname, callback)
        elif S_ISREG(mode):
            # It's a file, call the callback function
            callback(pathname)
        else:
            # Unknown file type, print a message
            print('Skipping %s' % pathname)

def visitfile(file):
    print('visiting', file)

if __name__ == '__main__':
    walktree(sys.argv[1], visitfile)
```

An additional utility function is provided to convert a file's mode in a human readable string :

`stat.filemode(mode)`

Convert a file's mode to a string of the form '-rwxrwxrwx'.

Ajouté dans la version 3.3.

Modifié dans la version 3.4 : The function supports `S_IFDOOR`, `S_IFPORT` and `S_IFWHT`.

All the variables below are simply symbolic indexes into the 10-tuple returned by `os.stat()`, `os.fstat()` or `os.lstat()`.

`stat.ST_MODE`

Mode de protection de l'*inode*.

`stat.ST_INO`

Numéro d'*inode*.

`stat.ST_DEV`

Device inode resides on.

`stat.ST_NLINK`

Nombre de liens vers l'*inode*.

`stat.ST_UID`

Identifiant utilisateur du propriétaire.

`stat.ST_GID`

Identifiant de groupe du propriétaire.

`stat.ST_SIZE`

Size in bytes of a plain file ; amount of data waiting on some special files.

`stat.ST_ATIME`

L'heure du dernier accès.

stat.**ST_MTIME**

L'heure de la dernière modification.

stat.**ST_CTIME**

The "ctime" as reported by the operating system. On some systems (like Unix) is the time of the last metadata change, and, on others (like Windows), is the creation time (see platform documentation for details).

The interpretation of "file size" changes according to the file type. For plain files this is the size of the file in bytes. For FIFOs and sockets under most flavors of Unix (including Linux in particular), the "size" is the number of bytes waiting to be read at the time of the call to `os.stat()`, `os.fstat()`, or `os.lstat()` ; this can sometimes be useful, especially for polling one of these special files after a non-blocking open. The meaning of the size field for other character and block devices varies more, depending on the implementation of the underlying system call.

The variables below define the flags used in the `ST_MODE` field.

Use of the functions above is more portable than use of the first set of flags :

stat.**S_IFSOCK**

Socket.

stat.**S_IFLNK**

Lien symbolique.

stat.**S_IFREG**

Fichier normal.

stat.**S_IFBLK**

Périphérique en mode bloc.

stat.**S_IFDIR**

Dossier.

stat.**S_IFCHR**

Périphérique en mode caractère.

stat.**S_IFIFO**

FIFO.

stat.**S_IFDOOR**

Door.

Ajouté dans la version 3.4.

stat.**S_IFPORT**

Event port.

Ajouté dans la version 3.4.

stat.**S_IFWHT**

Whiteout.

Ajouté dans la version 3.4.

Note

`S_IFDOOR`, `S_IFPORT` or `S_IFWHT` sont définies à 0 si la plateforme ne prend pas en charge les types de fichiers.

Les drapeaux suivant peuvent aussi être utilisé dans l'argument *mode* de `os.chmod()` :

stat.**S_ISUID**

Définit le bit UID.

stat.**S_ISGID**

Set-group-ID bit. This bit has several special uses. For a directory it indicates that BSD semantics is to be used for that directory : files created there inherit their group ID from the directory, not from the effective group ID of the creating process, and directories created there will also get the *S_ISGID* bit set. For a file that does not have the group execution bit (*S_IXGRP*) set, the set-group-ID bit indicates mandatory file/record locking (see also *S_ENFMT*).

stat.**S_ISVTX**

Sticky bit. When this bit is set on a directory it means that a file in that directory can be renamed or deleted only by the owner of the file, by the owner of the directory, or by a privileged process.

stat.**S_IRWXU**

Mask for file owner permissions.

stat.**S_IRUSR**

Le propriétaire possède le droit de lecture.

stat.**S_IWUSR**

Le propriétaire possède le droit d'écriture.

stat.**S_IXUSR**

Le propriétaire possède le droit d'exécution.

stat.**S_IRWXG**

Masque pour extraire les permissions du groupe.

stat.**S_IRGRP**

Le groupe possède le droit de lecture.

stat.**S_IWGRP**

Le groupe possède le droit d'écriture.

stat.**S_IXGRP**

Le groupe possède le droit d'exécution.

stat.**S_IRWXO**

Mask for permissions for others (not in group).

stat.**S_IROTH**

Others have read permission.

stat.**S_IWOTH**

Others have write permission.

stat.**S_IXOTH**

Others have execute permission.

stat.**S_ENFMT**

System V file locking enforcement. This flag is shared with *S_ISGID* : file/record locking is enforced on files that do not have the group execution bit (*S_IXGRP*) set.

stat.**S_IREAD**

Unix V7 synonym for *S_IRUSR*.

stat.**S_IWRITE**

Unix V7 synonym for *S_IWUSR*.

stat.**S_IEXEC**

Unix V7 synonym for *S_IXUSR*.

The following flags can be used in the *flags* argument of *os.chflags()* :

`stat.UF_SETTABLE`
All user settable flags.

Ajouté dans la version 3.13.

`stat.UF_NODUMP`
Do not dump the file.

`stat.UF_IMMUTABLE`
Le fichier ne peut pas être modifié.

`stat.UF_APPEND`
The file may only be appended to.

`stat.UF_OPAQUE`
The directory is opaque when viewed through a union stack.

`stat.UF_NOUNLINK`
Le fichier ne peut pas être renommé ou supprimé.

`stat.UF_COMPRESSED`
The file is stored compressed (macOS 10.6+).

`stat.UF_TRACKED`
Used for handling document IDs (macOS)

Ajouté dans la version 3.13.

`stat.UF_DATAVAULT`
The file needs an entitlement for reading or writing (macOS 10.13+)

Ajouté dans la version 3.13.

`stat.UF_HIDDEN`
Le fichier ne peut pas être affiché dans une GUI (macOS 10.5+).

`stat.SF_SETTABLE`
All super-user changeable flags

Ajouté dans la version 3.13.

`stat.SF_SUPPORTED`
All super-user supported flags

Availability : macOS

Ajouté dans la version 3.13.

`stat.SF_SYNTHETIC`
All super-user read-only synthetic flags

Availability : macOS

Ajouté dans la version 3.13.

`stat.SF_ARCHIVED`
Le fichier peut être archivé.

`stat.SF_IMMUTABLE`
Le fichier ne peut pas être modifié.

`stat.SF_APPEND`
The file may only be appended to.

stat.**SF_RESTRICTED**

The file needs an entitlement to write to (macOS 10.13+)

Ajouté dans la version 3.13.

stat.**SF_NOUNLINK**

Le fichier ne peut pas être renommé ou supprimé.

stat.**SF_SNAPSHOT**

The file is a snapshot file.

stat.**SF_FIRMLINK**

The file is a firmlink (macOS 10.15+)

Ajouté dans la version 3.13.

stat.**SF_DATALESS**

The file is a dataless object (macOS 10.15+)

Ajouté dans la version 3.13.

See the *BSD or macOS systems man page *chflags(2)* for more information.

On Windows, the following file attribute constants are available for use when testing bits in the `st_file_attributes` member returned by *os.stat()*. See the Windows API documentation for more detail on the meaning of these constants.

stat.**FILE_ATTRIBUTE_ARCHIVE**

stat.**FILE_ATTRIBUTE_COMPRESSED**

stat.**FILE_ATTRIBUTE_DEVICE**

stat.**FILE_ATTRIBUTE_DIRECTORY**

stat.**FILE_ATTRIBUTE_ENCRYPTED**

stat.**FILE_ATTRIBUTE_HIDDEN**

stat.**FILE_ATTRIBUTE_INTEGRITY_STREAM**

stat.**FILE_ATTRIBUTE_NORMAL**

stat.**FILE_ATTRIBUTE_NOT_CONTENT_INDEXED**

stat.**FILE_ATTRIBUTE_NO_SCRUB_DATA**

stat.**FILE_ATTRIBUTE_OFFLINE**

stat.**FILE_ATTRIBUTE_READONLY**

stat.**FILE_ATTRIBUTE_REPARSE_POINT**

stat.**FILE_ATTRIBUTE_SPARSE_FILE**

stat.**FILE_ATTRIBUTE_SYSTEM**

stat.**FILE_ATTRIBUTE_TEMPORARY**

stat.**FILE_ATTRIBUTE_VIRTUAL**

Ajouté dans la version 3.5.

On Windows, the following constants are available for comparing against the `st_reparse_tag` member returned by *os.lstat()*. These are well-known constants, but are not an exhaustive list.

stat.**IO_REPARSE_TAG_SYMLINK**

stat.**IO_REPARSE_TAG_MOUNT_POINT**

stat.**IO_REPARSE_TAG_APPEXECLINK**

Ajouté dans la version 3.8.

11.4 filecmp --- File and Directory Comparisons

Code source : Lib/filecmp.py

Le module *filecmp* définit les fonctions permettant de comparer les fichiers et les répertoires, avec différents compromis optionnels durée / exactitude. Pour comparer des fichiers, voir aussi le module *difflib*.

Le module *filecmp* définit les fonctions suivantes :

filecmp.**cmp**(*f1*, *f2*, *shallow=True*)

Compare les fichiers nommés *f1* et *f2* , renvoie `True` s'ils semblent égaux, `False` sinon.

If *shallow* is true and the *os.stat()* signatures (file type, size, and modification time) of both files are identical, the files are taken to be equal.

Otherwise, the files are treated as different if their sizes or contents differ.

Notez qu'aucun programme externe n'est appelé à partir de cette fonction, ce qui lui confère des qualités de portabilité et d'efficacité.

Cette fonction utilise un cache pour les comparaisons antérieures et les résultats, les entrées du cache étant invalidées si les informations *os.stat()* du fichier sont modifiées. La totalité du cache peut être effacée avec *clear_cache()*.

filecmp.**cmpfiles**(*dir1*, *dir2*, *common*, *shallow=True*)

Compare les fichiers des deux répertoires *dir1* et *dir2* dont les noms sont donnés par *common*.

Renvoie trois listes de noms de fichiers : *match* , *mismatch*, *errors*. *match* contient la liste des fichiers qui correspondent, *mismatch* contient les noms de ceux qui ne correspondent pas et *errors* répertorie les noms des fichiers qui n'ont pas pu être comparés. Les fichiers sont répertoriés dans *errors* s'ils n'existent pas dans l'un des répertoires, si l'utilisateur ne dispose pas de l'autorisation nécessaire pour les lire ou si la comparaison n'a pas pu être effectuée pour une autre raison.

Le paramètre *shallow* a la même signification et la même valeur par défaut que pour *filecmp.cmp()*.

Par exemple, `cmpfiles('a', 'b', ['c', 'd/e'])` compare `a/c` et `b/c` et `a/d/e` avec `b/d/e`. `'c'` et `'d/e'` seront chacun dans l'une des trois listes renvoyées.

filecmp.**clear_cache**()

Efface le cache *filecmp*. Cela peut être utile si un fichier est comparé juste après avoir été modifié (dans un délai inférieur à la résolution *mtime* du système de fichiers sous-jacent).

Ajouté dans la version 3.4.

11.4.1 La classe dircmp

class filecmp.**dircmp**(*a*, *b*, *ignore=None*, *hide=None*, *, *shallow=True*)

Construct a new directory comparison object, to compare the directories *a* and *b*. *ignore* is a list of names to ignore, and defaults to *filecmp.DEFAULT_IGNORES*. *hide* is a list of names to hide, and defaults to `[os.curdir, os.pardir]`.

The *dircmp* class compares files by doing *shallow* comparisons as described for *filecmp.cmp()* by default using the *shallow* parameter.

Modifié dans la version 3.13 : Added the *shallow* parameter.

La classe *dircmp* fournit les méthodes suivantes :

report()

Affiche (sur *sys.stdout*) une comparaison entre *a* et *b*.

report_partial_closure()

Affiche une comparaison entre *a* et *b* et les sous-répertoires immédiats communs.

report_full_closure()

Affiche une comparaison entre a et b et les sous-répertoires communs (récursivement).

La classe `dircmp` offre un certain nombre d'attributs intéressants qui peuvent être utilisés pour obtenir diverses informations sur les arborescences de répertoires comparées.

Note that via `__getattr__()` hooks, all attributes are computed lazily, so there is no speed penalty if only those attributes which are lightweight to compute are used.

left

Le répertoire *a*.

right

Le répertoire *b*.

left_list

Fichiers et sous-répertoires dans *a* , filtrés par *hide* et *ignore*.

right_list

Fichiers et sous-répertoires dans *b* , filtrés par *hide* et *ignore*.

common

Fichiers et sous-répertoires à la fois dans *a* et *b*.

left_only

Fichiers et sous-répertoires uniquement dans *a*.

right_only

Fichiers et sous-répertoires uniquement dans *b*.

common_dirs

Sous-répertoires à la fois dans *a* et *b*.

common_files

Fichiers à la fois dans *a* et *b*.

common_funny

Noms dans *a* et *b* , tels que le type diffère entre les répertoires, ou noms pour lesquels `os.stat()` signale une erreur.

same_files

Fichiers identiques dans *a* et *b*, en utilisant l'opérateur de comparaison de fichiers de la classe.

diff_files

Fichiers figurant à la fois dans *a* et dans *b* , dont le contenu diffère en fonction de l'opérateur de comparaison de fichiers de la classe.

funny_files

Fichiers à la fois dans *a* et dans *b* , mais ne pouvant pas être comparés.

subdirs

A dictionary mapping names in `common_dirs` to `dircmp` instances (or MyDirCmp instances if this instance is of type MyDirCmp, a subclass of `dircmp`).

Modifié dans la version 3.10 : Previously entries were always `dircmp` instances. Now entries are the same type as *self*, if *self* is a subclass of `dircmp`.

`filecmp.`**DEFAULT_IGNORES**

Ajouté dans la version 3.4.

Liste des répertoires ignorés par défaut par `dircmp`.

Voici un exemple simplifié d'utilisation de l'attribut `subdirs` pour effectuer une recherche récursive dans deux répertoires afin d'afficher des fichiers communs différents :

```
>>> from filecmp import dircmp
>>> def print_diff_files(dcmp):
...     for name in dcmp.diff_files:
...         print("diff_file %s found in %s and %s" % (name, dcmp.left,
...               dcmp.right))
...     for sub_dcmp in dcmp.subdirs.values():
...         print_diff_files(sub_dcmp)
...
>>> dcmp = dircmp('dir1', 'dir2')
>>> print_diff_files(dcmp)
```

11.5 `tempfile` --- Generate temporary files and directories

Code source : Lib/tempfile.py

This module creates temporary files and directories. It works on all supported platforms. `TemporaryFile`, `NamedTemporaryFile`, `TemporaryDirectory`, and `SpooledTemporaryFile` are high-level interfaces which provide automatic cleanup and can be used as *context managers*. `mkstemp()` and `mkdtemp()` are lower-level functions which require manual cleanup.

Toutes les fonctions et constructeurs appelables par l'utilisateur ont des arguments additionnels qui permettent de contrôler directement le chemin et le nom des répertoires et fichiers. Les noms de fichiers utilisés par ce module incluent une chaîne de caractères aléatoires qui leur permet d'être créés de manière sécurisée dans des répertoires temporaires partagés. Afin de maintenir la compatibilité descendante, l'ordre des arguments est quelque peu étrange ; pour des questions de clarté, il est recommandé d'utiliser les arguments nommés.

Le module définit les éléments suivants pouvant être appelés par l'utilisateur :

`tempfile.`**`TemporaryFile`**(*mode='w+b'*, *buffering=-1*, *encoding=None*, *newline=None*, *suffix=None*, *prefix=None*, *dir=None*, **, errors=None*)

Renvoie un *objet fichier* qui peut être utilisé comme une zone de stockage temporaire. Le fichier est créé de manière sécurisée, utilisant les mêmes règles que `mkstemp()`. Il sera détruit dès qu'il sera fermé (y compris lorsque le fichier est implicitement fermé quand il est collecté par le ramasse-miettes). Sous Unix, soit l'entrée du répertoire n'est pas du tout créée, soit elle est supprimée immédiatement après la création du fichier. Les autres plateformes ne gèrent pas cela, votre code ne doit pas compter sur un fichier temporaire créé en utilisant cette fonction ayant ou non un nom visible sur le système de fichiers.

The resulting object can be used as a *context manager* (see *Exemples*). On completion of the context or destruction of the file object the temporary file will be removed from the filesystem.

Le paramètre *mode* vaut par défaut `'w+b'` afin que le fichier créé puisse être lu et écrit sans être fermé. Le mode binaire est utilisé afin que le comportement soit le même sur toutes les plateformes quelle que soit la donnée qui est stockée. *buffering*, *encoding*, *errors* et *newline* sont interprétés de la même façon que pour `open()`.

Les paramètres *dir*, *prefix* et *suffix* ont la même signification et les mêmes valeurs par défaut que dans `mkstemp()`.

L'objet renvoyé est un véritable fichier sur les plateformes POSIX. Sur les autres plateformes, c'est un objet fichier-compatible, dont l'attribut `file` donne le véritable fichier.

The `os.O_TMPFILE` flag is used if it is available and works (Linux-specific, requires Linux kernel 3.11 or later).

Sur les plateformes qui ne sont pas POSIX ni Cygwin, `TemporaryFile` est un alias de `NamedTemporaryFile`.

Lève un *évènement d'audit* `tempfile.mkstemp` avec comme argument `fullpath`.

Modifié dans la version 3.5 : The `os.O_TMPFILE` flag is now used if available.

Modifié dans la version 3.8 : Le paramètre *errors* a été ajouté.

tempfile.**NamedTemporaryFile**(*mode='w+b'*, *buffering=-1*, *encoding=None*, *newline=None*, *suffix=None*, *prefix=None*, *dir=None*, *delete=True*, ***, *errors=None*, *delete_on_close=True*)

This function operates exactly as `TemporaryFile()` does, except the following differences :

— This function returns a file that is guaranteed to have a visible name in the file system.

— To manage the named file, it extends the parameters of `TemporaryFile()` with *delete* and *delete_on_close* parameters that determine whether and how the named file should be automatically deleted.

The returned object is always a *file-like object* whose `file` attribute is the underlying true file object. This file-like object can be used in a `with` statement, just like a normal file. The name of the temporary file can be retrieved from the `name` attribute of the returned file-like object. On Unix, unlike with the `TemporaryFile()`, the directory entry does not get unlinked immediately after the file creation.

If *delete* is true (the default) and *delete_on_close* is true (the default), the file is deleted as soon as it is closed. If *delete* is true and *delete_on_close* is false, the file is deleted on context manager exit only, or else when the *file-like object* is finalized. Deletion is not always guaranteed in this case (see `object.__del__()`). If *delete* is false, the value of *delete_on_close* is ignored.

Therefore to use the name of the temporary file to reopen the file after closing it, either make sure not to delete the file upon closure (set the *delete* parameter to be false) or, in case the temporary file is created in a `with` statement, set the *delete_on_close* parameter to be false. The latter approach is recommended as it provides assistance in automatic cleaning of the temporary file upon the context manager exit.

Opening the temporary file again by its name while it is still open works as follows :

— On POSIX the file can always be opened again.

— On Windows, make sure that at least one of the following conditions are fulfilled :

  — *delete* is false

  — additional open shares delete access (e.g. by calling `os.open()` with the flag `O_TEMPORARY`)

  — *delete* is true but *delete_on_close* is false. Note, that in this case the additional opens that do not share delete access (e.g. created via builtin `open()`) must be closed before exiting the context manager, else the `os.unlink()` call on context manager exit will fail with a `PermissionError`.

On Windows, if *delete_on_close* is false, and the file is created in a directory for which the user lacks delete access, then the `os.unlink()` call on exit of the context manager will fail with a `PermissionError`. This cannot happen when *delete_on_close* is true because delete access is requested by the open, which fails immediately if the requested access is not granted.

On POSIX (only), a process that is terminated abruptly with SIGKILL cannot automatically delete any NamedTemporaryFiles it created.

Lève un *évènement d'audit* `tempfile.mkstemp` avec comme argument `fullpath`.

Modifié dans la version 3.8 : Le paramètre *errors* a été ajouté.

Modifié dans la version 3.12 : Added *delete_on_close* parameter.

class tempfile.**SpooledTemporaryFile**(*max_size=0*, *mode='w+b'*, *buffering=-1*, *encoding=None*, *newline=None*, *suffix=None*, *prefix=None*, *dir=None*, ***, *errors=None*)

This class operates exactly as `TemporaryFile()` does, except that data is spooled in memory until the file size exceeds *max_size*, or until the file's `fileno()` method is called, at which point the contents are written to disk and operation proceeds as with `TemporaryFile()`.

rollover()

The resulting file has one additional method, `rollover()`, which causes the file to roll over to an on-disk file regardless of its size.

The returned object is a file-like object whose `_file` attribute is either an `io.BytesIO` or `io.TextIOWrapper` object (depending on whether binary or text *mode* was specified) or a true file object, depending on whether `rollover()` has been called. This file-like object can be used in a `with` statement, just like a normal file.

Modifié dans la version 3.3 : the truncate method now accepts a *size* argument.

Modifié dans la version 3.8 : Le paramètre *errors* a été ajouté.

Modifié dans la version 3.11 : Fully implements the `io.BufferedIOBase` and `io.TextIOBase` abstract

base classes (depending on whether binary or text *mode* was specified).

class tempfile.**TemporaryDirectory**(*suffix=None*, *prefix=None*, *dir=None*, *ignore_cleanup_errors=False*, *, *delete=True*)

This class securely creates a temporary directory using the same rules as `mkdtemp()`. The resulting object can be used as a *context manager* (see *Exemples*). On completion of the context or destruction of the temporary directory object, the newly created temporary directory and all its contents are removed from the filesystem.

name

The directory name can be retrieved from the `name` attribute of the returned object. When the returned object is used as a *context manager*, the `name` will be assigned to the target of the `as` clause in the `with` statement, if there is one.

cleanup()

The directory can be explicitly cleaned up by calling the `cleanup()` method. If *ignore_cleanup_errors* is true, any unhandled exceptions during explicit or implicit cleanup (such as a `PermissionError` removing open files on Windows) will be ignored, and the remaining removable items deleted on a "best-effort" basis. Otherwise, errors will be raised in whatever context cleanup occurs (the `cleanup()` call, exiting the context manager, when the object is garbage-collected or during interpreter shutdown).

The *delete* parameter can be used to disable cleanup of the directory tree upon exiting the context. While it may seem unusual for a context manager to disable the action taken when exiting the context, it can be useful during debugging or when you need your cleanup behavior to be conditional based on other logic.

Lève un *évènement d'audit* `tempfile.mkdtemp` avec comme argument `fullpath`.

Ajouté dans la version 3.2.

Modifié dans la version 3.10 : ajout du paramètre *ignore_cleanup_errors*.

Modifié dans la version 3.12 : Added the *delete* parameter.

tempfile.**mkstemp**(*suffix=None*, *prefix=None*, *dir=None*, *text=False*)

Crée un fichier temporaire de la manière la plus sécurisée qui soit. Il n'y a pas de problème d'accès concurrent (*race condition* en anglais) au moment de la création du fichier, en supposant que la plateforme implémente correctement l'option `os.O_EXCL` pour `os.open()`. Le fichier est seulement accessible en lecture et écriture par l'ID de l'utilisateur créateur. Si la plateforme utilise des bits de permissions pour indiquer si le fichier est exécutable, alors le fichier n'est exécutable par personne. Le descripteur de fichier n'est pas hérité par les processus fils.

À la différence de `TemporaryFile()`, l'utilisateur de `mkstemp()` est responsable de la suppression du fichier temporaire quand il n'en a plus besoin.

Si *suffix* ne vaut pas `None`, le nom de fichier se terminera avec ce suffixe, sinon il n'y aura pas de suffixe. `mkstemp()` ne met pas de point entre le nom du fichier et le suffixe. Si vous en avez besoin, mettez le point au début de *suffix*.

Si *prefix* ne vaut pas `None`, le nom de fichier commencera avec ce préfixe, sinon un préfixe par défaut est utilisé. La valeur par défaut est la valeur retournée par `gettempprefix()` ou `gettempprefixb()`.

Si *dir* ne vaut pas `None`, le fichier sera créé dans ce répertoire, autrement, un répertoire par défaut sera utilisé. Le répertoire par défaut est choisi depuis une liste dépendante de la plateforme, mais l'utilisateur de l'application peut contrôler l'emplacement du répertoire en spécifiant les variables d'environnement *TMPDIR*, *TEMP* ou *TMP*. Il n'y a pas de garantie que le nom de fichier généré aura de bonnes propriétés telles que ne pas avoir besoin de le mettre entre guillemets lorsque celui-ci est passé à des commandes externes via `os.popen()`.

Si l'un des paramètres *suffix*, *prefix* et *dir* n'est pas `None`, ils doivent être du même type. S'ils sont de type `bytes`, le nom renvoyé est de type `bytes` plutôt que de type `str`. Si vous voulez forcer la valeur renvoyée en `bytes`, passez `suffix=b''`.

Le fichier est ouvert en mode binaire, sauf si *text* est mis à vrai, auquel cas il est ouvert en mode textuel.

`mkstemp()` renvoie une paire contenant un descripteur (*handle* en anglais) au niveau du système d'exploitation vers un fichier ouvert (le même que renvoie `os.open()`) et le chemin d'accès absolu de ce fichier, dans cet ordre.

Lève un *évènement d'audit* `tempfile.mkstemp` avec comme argument `fullpath`.

Modifié dans la version 3.5 : *suffix*, *prefix*, et *dir* peuvent maintenant être spécifiés en `bytes` pour obtenir un résultat en `bytes`. Avant cela, le type `str` était le seul autorisé. *suffix* et *prefix* acceptent maintenant la valeur par défaut `None` pour que la valeur par défaut appropriée soit utilisée.

Modifié dans la version 3.6 : Le paramètre *dir* accepte un *objet fichier-compatible*.

`tempfile.`**`mkdtemp`**(*suffix=None*, *prefix=None*, *dir=None*)

Crée un répertoire temporaire de la manière la plus sécurisée qu'il soit. Il n'y a pas de problème d'accès concurrent (*race condition* en anglais) au moment de la création du répertoire. Le répertoire est accessible en lecture, en écriture, et son contenu lisible uniquement pour l'ID de l'utilisateur créateur.

L'utilisateur de `mkdtemp()` est responsable de la suppression du répertoire temporaire et de son contenu lorsqu'il n'en a plus besoin.

Les arguments *prefix*, *suffix*, et *dir* sont les mêmes que pour `mkstemp()`.

`mkdtemp()` renvoie le chemin absolu du nouveau répertoire.

Lève un *évènement d'audit* `tempfile.mkdtemp` avec comme argument `fullpath`.

Modifié dans la version 3.5 : *suffix*, *prefix*, et *dir* peuvent maintenant être spécifiés en `bytes` pour obtenir un résultat en `bytes`. Avant cela, le type `str` était le seul autorisé. *suffix* et *prefix* acceptent maintenant la valeur par défaut `None` pour que la valeur par défaut appropriée soit utilisée.

Modifié dans la version 3.6 : Le paramètre *dir* accepte un *objet fichier-compatible*.

Modifié dans la version 3.12 : `mkdtemp()` now always returns an absolute path, even if *dir* is relative.

`tempfile.`**`gettempdir`**()

Renvoie le nom du répertoire utilisé pour les fichiers temporaires. C'est la valeur par défaut pour l'argument *dir* de toutes les fonctions de ce module.

Python cherche un répertoire parmi une liste standard de répertoires dans lequel l'utilisateur final peut créer des fichiers. La liste est :

1. Le répertoire correspondant à la variable d'environnement `TMPDIR`.
2. Le répertoire correspondant à la variable d'environnement `TEMP`.
3. Le répertoire correspondant à la variable d'environnement `TMP`.
4. Un emplacement dépendant de la plateforme :
   — Sur Windows, les répertoires `C:\TEMP`, `C:\TMP`, `\TEMP`, et `\TMP`, dans cet ordre.
   — Sur toutes les autres plate-formes, les répertoires `/tmp`, `/var/tmp`, et `/usr/tmp`, dans cet ordre.
5. En dernier ressort, le répertoire de travail courant.

Le résultat de cette recherche est mis en cache, voir la description de `tempdir` dessous.

Modifié dans la version 3.10 : Cette fonction renvoie désormais systématiquement une chaîne de caractères. Auparavant, elle pouvait renvoyer la valeur de `tempdir` quel que soit son type tant que ce n'était pas `None`.

`tempfile.`**`gettempdirb`**()

Similaire à `gettempdir()` mais la valeur retournée est en *bytes*.

Ajouté dans la version 3.5.

`tempfile.`**`gettempprefix`**()

Renvoie le préfixe de nom de fichier utilisé pour créer les fichiers temporaires. Cela ne contient pas le nom du répertoire.

`tempfile.`**`gettempprefixb`**()

Similaire à `gettempprefix()` mais la valeur retournée est en *bytes*.

Ajouté dans la version 3.5.

Le module utilise une variable globale pour stocker le nom du répertoire utilisé pour les fichiers temporaires renvoyés par `gettempdir()`. Vous pouvez directement utiliser la variable globale pour surcharger le processus de sélection, mais ceci est déconseillé. Toutes les fonctions de ce module prennent un argument *dir* qui peut être utilisé pour

spécifier le répertoire. Il s'agit de la méthode recommandée car elle n'interfère pas avec le code extérieur en modifiant le comportement global du module.

`tempfile.tempdir`

Quand une valeur autre que `None` est spécifiée, cette variable définit la valeur par défaut pour l'argument *dir* des fonctions définies dans ce module, et en particulier son type (`bytes` ou `str`). Les *objets fichier-compatibles* ne sont pas acceptés.

Si `tempdir` vaut `None` (par défaut) pour n'importe laquelle des fonctions ci-dessus, sauf `gettempprefix()`, la variable est initialisée suivant l'algorithme décrit dans `gettempdir()`.

> **Note**
>
> Attention, le fait de mettre `tempdir` à une valeur de type `bytes` a l'effet pervers de changer globalement en `bytes` le type de retour de `mkstemp()` et `mkdtemp()` lorsqu'elles sont appelées sans qu'aucun des paramètres *prefix*, *suffix* et *dir* ne soit de type `str`. Ce comportement par défaut surprenant est préservé par souci de compatibilité avec l'implémentation historique du module. Il est fortement recommandé de ne pas s'y fier.

11.5.1 Exemples

Voici quelques exemples classiques d'utilisation du module `tempfile` :

```
>>> import tempfile

# create a temporary file and write some data to it
>>> fp = tempfile.TemporaryFile()
>>> fp.write(b'Hello world!')
# read data from file
>>> fp.seek(0)
>>> fp.read()
b'Hello world!'
# close the file, it will be removed
>>> fp.close()

# create a temporary file using a context manager
>>> with tempfile.TemporaryFile() as fp:
...     fp.write(b'Hello world!')
...     fp.seek(0)
...     fp.read()
b'Hello world!'
>>>
# file is now closed and removed

# create a temporary file using a context manager
# close the file, use the name to open the file again
>>> with tempfile.NamedTemporaryFile(delete_on_close=False) as fp:
...     fp.write(b'Hello world!')
...     fp.close()
... # the file is closed, but not removed
... # open the file again by using its name
...     with open(fp.name, mode='rb') as f:
...         f.read()
b'Hello world!'
>>>
# file is now removed
```

(suite sur la page suivante)

(suite de la page précédente)

```
# create a temporary directory using the context manager
>>> with tempfile.TemporaryDirectory() as tmpdirname:
...     print('created temporary directory', tmpdirname)
>>>
# directory and contents have been removed
```

11.5.2 Fonctions et variables obsolètes

Historiquement, la méthode pour créer des fichiers temporaires consistait à générer un nom de fichier avec la fonction `mktemp()` puis créer un fichier en utilisant ce nom. Malheureusement, cette méthode n'est pas fiable car un autre processus peut créer un fichier avec ce nom entre l'appel à la fonction `mktemp()` et la tentative de création de fichier par le premier processus en cours. La solution est de combiner les deux étapes et de créer le fichier immédiatement. Cette approche est utilisée par `mkstemp()` et les autres fonctions décrites plus haut.

`tempfile.`**`mktemp`**(*suffix=''*, *prefix='tmp'*, *dir=None*)

Obsolète depuis la version 2.3 : Utilisez `mkstemp()` à la place.

Renvoie le chemin absolu d'un fichier qui n'existe pas lorsque l'appel est fait. Les arguments *prefix*, *suffix*, et *dir* sont similaires à ceux de `mkstemp()` mais les noms de fichiers en *bytes*, `sufix=None` et `prefix=None` ne sont pas implémentées.

Avertissement

Utiliser cette fonction peut introduire une faille de sécurité dans votre programme. Avant que vous n'ayez le temps de faire quoi que ce soit avec le nom de fichier renvoyé, quelqu'un peut l'utiliser. L'utilisation de `mktemp()` peut être remplacée facilement avec `NamedTemporaryFile()` en y passant le paramètre `delete=False` :

```
>>> f = NamedTemporaryFile(delete=False)
>>> f.name
'/tmp/tmptjujjt'
>>> f.write(b"Hello World!\n")
13
>>> f.close()
>>> os.unlink(f.name)
>>> os.path.exists(f.name)
False
```

11.6 `glob` --- Unix style pathname pattern expansion

Code source : Lib/glob.py

The `glob` module finds all the pathnames matching a specified pattern according to the rules used by the Unix shell, although results are returned in arbitrary order. No tilde expansion is done, but `*`, `?`, and character ranges expressed with `[]` will be correctly matched. This is done by using the `os.scandir()` and `fnmatch.fnmatch()` functions in concert, and not by actually invoking a subshell.

Note that files beginning with a dot (`.`) can only be matched by patterns that also start with a dot, unlike `fnmatch.fnmatch()` or `pathlib.Path.glob()`. (For tilde and shell variable expansion, use `os.path.expanduser()` and `os.path.expandvars()`.)

Pour une correspondance littérale, il faut entourer le métacaractère par des crochets. Par exemple, `'[?]'` reconnaît le caractère `'?'`.

The `glob` module defines the following functions :

`glob.glob`(*pathname*, *, *root_dir=None*, *dir_fd=None*, *recursive=False*, *include_hidden=False*)

Return a possibly empty list of path names that match *pathname*, which must be a string containing a path specification. *pathname* can be either absolute (like `/usr/src/Python-1.5/Makefile`) or relative (like `../../Tools/*/*.gif`), and can contain shell-style wildcards. Broken symlinks are included in the results (as in the shell). Whether or not the results are sorted depends on the file system. If a file that satisfies conditions is removed or added during the call of this function, whether a path name for that file will be included is unspecified.

Si *root_dir* n'est pas `None`, cela doit être un *objet simili-chemin* spécifiant le dossier racine de la recherche. Cela a le même effet sur `glob()` que de changer le dossier courant avant l'appel de la fonction. Si *pathname* est relatif, les chemins du résultat seront relatifs au *root_dir* .

Cette fonction prend en charge les *chemins relatifs aux descripteurs de dossier* avec le paramètre *dir_fd*.

Si *recursive* est vrai, le motif "`**`" reconnaît tous les fichiers, aucun ou plusieurs répertoires, sous-répertoires et liens symboliques aux répertoires. Si le motif est suivi par `os.sep` ou `os.altsep` alors les fichiers ne sont pas inclus dans le résultat.

If *include_hidden* is true, "`**`" pattern will match hidden directories.

Lève un *évènement d'audit* `glob.glob` avec les arguments `pathname`, `recursive`.

Lève un *évènement d'audit* `glob.glob/2` avec les arguments `pathname`, `recursive`, `root_dir`, `dir_fd`.

> **Note**
>
> Utiliser le motif "`**`" dans de grandes arborescences de dossier peut consommer une quantité de temps démesurée.

> **Note**
>
> This function may return duplicate path names if *pathname* contains multiple "`**`" patterns and *recursive* is true.

Modifié dans la version 3.5 : Prise en charge des chemins récursifs utilisant le motif "`**`".

Modifié dans la version 3.10 : Paramètres *root_dir* et *dir_fd* ajoutés.

Modifié dans la version 3.11 : Added the *include_hidden* parameter.

`glob.iglob`(*pathname*, *, *root_dir=None*, *dir_fd=None*, *recursive=False*, *include_hidden=False*)

Renvoie un *itérateur* qui produit les mêmes valeurs que `glob()`, sans toutes les charger en mémoire simultanément.

Lève un *évènement d'audit* `glob.glob` avec les arguments `pathname`, `recursive`.

Lève un *évènement d'audit* `glob.glob/2` avec les arguments `pathname`, `recursive`, `root_dir`, `dir_fd`.

> **Note**
>
> This function may return duplicate path names if *pathname* contains multiple "`**`" patterns and *recursive* is true.

Modifié dans la version 3.5 : Prise en charge des chemins récursifs utilisant le motif "`**`".

Modifié dans la version 3.10 : Paramètres *root_dir* et *dir_fd* ajoutés.

Modifié dans la version 3.11 : Added the *include_hidden* parameter.

`glob.escape`(*pathname*)

Échappe tous les caractères spéciaux (`'?'`, `'*'` et `'['`). Cela est utile pour reconnaître une chaîne de caractère

littérale arbitraire qui contiendrait ce type de caractères. Les caractères spéciaux dans les disques et répertoires partagés (chemins UNC) ne sont pas échappés, e.g. sous Windows `escape('//?/c:/Quo vadis?.txt')` renvoie `'//?/c:/Quo vadis[?].txt'`.

Ajouté dans la version 3.4.

`glob.translate`(*pathname*, *, *recursive=False*, *include_hidden=False*, *seps=None*)

Convert the given path specification to a regular expression for use with `re.match()`. The path specification can contain shell-style wildcards.

Par exemple :

```
>>> import glob, re
>>>
>>> regex = glob.translate('**/*.txt', recursive=True, include_hidden=True)
>>> regex
'(?s:(?:.+/)?[^/]*\\.txt)\\Z'
>>> reobj = re.compile(regex)
>>> reobj.match('foo/bar/baz.txt')
<re.Match object; span=(0, 15), match='foo/bar/baz.txt'>
```

Path separators and segments are meaningful to this function, unlike `fnmatch.translate()`. By default wildcards do not match path separators, and `*` pattern segments match precisely one path segment.

If *recursive* is true, the pattern segment "`**`" will match any number of path segments.

If *include_hidden* is true, wildcards can match path segments that start with a dot (`.`).

A sequence of path separators may be supplied to the *seps* argument. If not given, `os.sep` and `altsep` (if available) are used.

Voir aussi

`pathlib.PurePath.full_match()` and `pathlib.Path.glob()` methods, which call this function to implement pattern matching and globbing.

Ajouté dans la version 3.13.

11.6.1 Exemples

Consider a directory containing the following files : `1.gif`, `2.txt`, `card.gif` and a subdirectory `sub` which contains only the file `3.txt`. `glob()` will produce the following results. Notice how any leading components of the path are preserved.

```
>>> import glob
>>> glob.glob('./[0-9].*')
['./1.gif', './2.txt']
>>> glob.glob('*.gif')
['1.gif', 'card.gif']
>>> glob.glob('?.gif')
['1.gif']
>>> glob.glob('**/*.txt', recursive=True)
['2.txt', 'sub/3.txt']
>>> glob.glob('./**/', recursive=True)
['./', './sub/']
```

Si le répertoire contient des fichiers commençant par `.`, ils ne sont pas reconnus par défaut. Par exemple, considérons un répertoire contenant `card.gif` et `.card.gif` :

```
>>> import glob
>>> glob.glob('*.gif')
['card.gif']
>>> glob.glob('.c*')
['.card.gif']
```

Voir aussi

The *fnmatch* module offers shell-style filename (not path) expansion.

Voir aussi

Le module *pathlib* offre une représentation objet de haut niveau des chemins.

11.7 fnmatch --- Unix filename pattern matching

Code source : Lib/fnmatch.py

Ce module fournit la gestion des caractères de remplacement de style shell Unix, qui ne sont *pas* identiques à ceux utilisés dans les expressions régulières (documentés dans le module *re*). Les caractères spéciaux utilisés comme caractères de remplacement de style shell sont :

Motif	Signification
`*`	reconnaît n'importe quoi
`?`	reconnaît n'importe quel caractère unique
`[seq]`	reconnaît n'importe quel caractère dans *seq*
`[!seq]`	reconnaît n'importe quel caractère qui n'est pas dans *seq*

Pour une correspondance littérale, il faut entourer le métacaractère par des crochets. Par exemple, `'[?]'` reconnaît le caractère `'?'`.

Notons que le séparateur de nom de fichiers (`'/'` sous Unix) n'est *pas* traité de manière spéciale par ce module. Voir le module *glob* pour la recherche de chemins (*glob* utilise *filter()* pour reconnaître les composants d'un chemin). De la même manière, les noms de fichiers commençant par un point ne sont pas traités de manière spéciale par ce module, et sont reconnus par les motifs `*` et `?`.

Unless stated otherwise, "filename string" and "pattern string" either refer to *str* or `ISO-8859-1` encoded *bytes* objects. Note that the functions documented below do not allow to mix a `bytes` pattern with a `str` filename, and vice-versa.

Finally, note that *functools.lru_cache()* with a *maxsize* of 32768 is used to cache the (typed) compiled regex patterns in the following functions : *fnmatch()*, *fnmatchcase()*, *filter()*.

`fnmatch.`**`fnmatch`**(*name*, *pat*)

Test whether the filename string *name* matches the pattern string *pat*, returning `True` or `False`. Both parameters are case-normalized using *os.path.normcase()*. *fnmatchcase()* can be used to perform a case-sensitive comparison, regardless of whether that's standard for the operating system.

Cet exemple affiche tous les noms de fichiers du répertoire courant ayant pour extension `.txt` :

```
import fnmatch
import os
```

(suite sur la page suivante)

(suite de la page précédente)

```
for file in os.listdir('.'):
    if fnmatch.fnmatch(file, '*.txt'):
        print(file)
```

`fnmatch.fnmatchcase(name, pat)`

Test whether the filename string *name* matches the pattern string *pat*, returning `True` or `False` ; the comparison is case-sensitive and does not apply `os.path.normcase()`.

`fnmatch.filter(names, pat)`

Construct a list from those elements of the *iterable* of filename strings *names* that match the pattern string *pat*. It is the same as `[n for n in names if fnmatch(n, pat)]`, but implemented more efficiently.

`fnmatch.translate(pat)`

Return the shell-style pattern *pat* converted to a regular expression for using with `re.match()`. The pattern is expected to be a `str`.

Exemple :

```
>>> import fnmatch, re
>>>
>>> regex = fnmatch.translate('*.txt')
>>> regex
'(?s:.*\\.txt)\\Z'
>>> reobj = re.compile(regex)
>>> reobj.match('foobar.txt')
<re.Match object; span=(0, 10), match='foobar.txt'>
```

Voir aussi

Module `glob`
Recherche de chemins de style shell Unix.

11.8 `linecache` --- Random access to text lines

Code source : Lib/linecache.py

Le module `linecache` permet d'obtenir n'importe quelle ligne d'un fichier source Python. Le cas classique où de nombreuses lignes sont accédées est optimisé en utilisant un cache interne. C'est utilisé par le module `traceback` pour récupérer les lignes à afficher dans les piles d'appels.

Les fichiers sont ouverts par la fonction `tokenize.open()`. Cette fonction utilise `tokenize.detect_encoding()` pour obtenir l'encodage du fichier. En l'absence d'un BOM et d'un cookie d'encodage, c'est l'encodage UTF-8 qui sera utilisé.

Le module `linecache` définit les fonctions suivantes :

`linecache.getline(filename, lineno, module_globals=None)`

Récupère la ligne *lineno* du fichier *filename*. Cette fonction ne lèvera jamais d'exception, elle préfèrera renvoyer `''` en cas d'erreur (le caractère de retour à la ligne sera inclus pour les lignes existantes).

Si le fichier *filename* n'existe pas, cette fonction vérifie d'abord la présence d'un chargeur `__loader__` dans *module_globals*, comme le prescrit la **PEP 302**. Si un chargeur est trouvé avec une méthode `get_source`, c'est cette méthode qui détermine les lignes sources (si `get_source()` renvoie `None`, cette valeur est remplacée par la chaîne vide `''`). Sinon, si le nom de fichier *filename* est relatif, il est recherché dans les dossiers du chemin de recherche des modules, `sys.path`.

`linecache.clearcache()`

Nettoie le cache. Utilisez cette fonction si vous n'avez plus besoin des lignes des fichiers précédemment lus via `getline()`.

`linecache.checkcache(filename=None)`

Vérifie la validité du cache. Utilisez cette fonction si les fichiers du cache pourraient avoir changé sur le disque, et que vous en voudriez une version à jour. Sans *filename*, toutes les entrées du cache seront vérifiées.

`linecache.lazycache(filename, module_globals)`

Récupère suffisamment d'informations sur un module situé hors du système de fichiers pour récupérer ses lignes plus tard via `getline()`, même si *module_globals* devient `None`. Cela évite de lire le fichier avant d'avoir besoin d'une ligne, tout en évitant de conserver les globales du module indéfiniment.

Ajouté dans la version 3.5.

Exemple :

```
>>> import linecache
>>> linecache.getline(linecache.__file__, 8)
'import sys\n'
```

11.9 `shutil` --- High-level file operations

Code source : Lib/shutil.py

Le module `shutil` propose des opérations de haut niveau sur les fichiers et ensembles de fichiers. En particulier, des fonctions pour copier et déplacer les fichiers sont proposées. Pour les opérations individuelles sur les fichiers, reportez-vous au module `os`.

> **Avertissement**
>
> Même les fonctions de copie haut niveau (`shutil.copy()`, `shutil.copy2()`) ne peuvent copier toutes les métadonnées des fichiers.
>
> Sur les plateformes POSIX, cela signifie que le propriétaire et le groupe du fichier sont perdus, ainsi que les *ACLs*. Sur Mac OS, le clonage de ressource et autres métadonnées ne sont pas utilisés. Cela signifie que les ressources seront perdues et que le type de fichier et les codes créateur ne seront pas corrects. Sur Windows, les propriétaires des fichiers, *ACLs* et flux de données alternatifs ne sont pas copiés.

11.9.1 Opérations sur les répertoires et les fichiers

`shutil.copyfileobj(fsrc, fdst[, length])`

Copy the contents of the *file-like object* *fsrc* to the file-like object *fdst*. The integer *length*, if given, is the buffer size. In particular, a negative *length* value means to copy the data without looping over the source data in chunks; by default the data is read in chunks to avoid uncontrolled memory consumption. Note that if the current file position of the *fsrc* object is not 0, only the contents from the current file position to the end of the file will be copied.

`shutil.copyfile(src, dst, *, follow_symlinks=True)`

Copy the contents (no metadata) of the file named *src* to a file named *dst* and return *dst* in the most efficient way possible. *src* and *dst* are *path-like objects* or path names given as strings.

dst doit être le nom complet du fichier cible; consultez `copy()` pour une fonction de copie qui accepte un chemin de répertoire cible. Si *src* et *dst* spécifient le même fichier, lève `SameFileError`.

La cible doit être accessible en écriture, sinon l'exception `OSError` est levée. Si *dst* existe déjà, il est remplacé. Les fichiers spéciaux comme les périphériques caractères ou bloc ainsi que les tubes (*pipes*) ne peuvent pas être copiés avec cette fonction.

Si *follow_symlinks* est faux et *src* est un lien symbolique, un nouveau lien symbolique est créé au lieu de copier le fichier pointé par *src*.

Lève un *événement d'audit* `shutil.copyfile` avec les arguments `src`, `dst`.

Modifié dans la version 3.3 : `IOError` était levée au lieu de `OSError`. Ajout de l'argument *follow_symlinks*. Maintenant renvoie *dst*.

Modifié dans la version 3.4 : Lève `SameFileError` au lieu de `Error`. Puisque la première est une sous-classe de la seconde, le changement assure la rétrocompatibilité.

Modifié dans la version 3.8 : Les appels système de copie rapide spécifiques à la plate-forme peuvent être utilisés en interne afin de copier le fichier plus efficacement. Voir la section *Platform-dependent efficient copy operations*.

exception `shutil.`**`SameFileError`**

Cette exception est levée si la source et la destination dans `copyfile()` sont le même fichier.

Ajouté dans la version 3.4.

`shutil.`**`copymode`**(*src*, *dst*, **, *follow_symlinks=True*)

Copy the permission bits from *src* to *dst*. The file contents, owner, and group are unaffected. *src* and *dst* are *path-like objects* or path names given as strings. If *follow_symlinks* is false, and both *src* and *dst* are symbolic links, `copymode()` will attempt to modify the mode of *dst* itself (rather than the file it points to). This functionality is not available on every platform ; please see `copystat()` for more information. If `copymode()` cannot modify symbolic links on the local platform, and it is asked to do so, it will do nothing and return.

Lève un *événement d'audit* `shutil.copymode` avec les arguments `src`, `dst`.

Modifié dans la version 3.3 : L'argument *follow_symlinks* a été ajouté.

`shutil.`**`copystat`**(*src*, *dst*, **, *follow_symlinks=True*)

Copy the permission bits, last access time, last modification time, and flags from *src* to *dst*. On Linux, `copystat()` also copies the "extended attributes" where possible. The file contents, owner, and group are unaffected. *src* and *dst* are *path-like objects* or path names given as strings.

Si *follow_symlinks* est faux et *src* et *dst* représentent des liens symboliques, `copystat()` agit sur les liens symboliques au lieu des fichiers cibles — elle lit les informations du lien symbolique de *src* et les écrit vers la destination pointée par *dst*.

> **Note**
>
> Toutes les plateformes n'offrent pas la possibilité d'examiner et modifier les liens symboliques. Python peut vous informer des fonctionnalités effectivement disponibles.
>
> — Si `os.chmod in os.supports_follow_symlinks` est `True`, `copystat()` peut modifier les octets de droits d'accès du lien symbolique.
> — If `os.utime in os.supports_follow_symlinks` is `True`, `copystat()` can modify the last access and modification times of a symbolic link.
> — If `os.chflags in os.supports_follow_symlinks` is `True`, `copystat()` can modify the flags of a symbolic link. (`os.chflags` is not available on all platforms.)
>
> On platforms where some or all of this functionality is unavailable, when asked to modify a symbolic link, `copystat()` will copy everything it can. `copystat()` never returns failure.
>
> Please see `os.supports_follow_symlinks` for more information.

Raises an *auditing event* `shutil.copystat` with arguments `src`, `dst`.

Modifié dans la version 3.3 : Added *follow_symlinks* argument and support for Linux extended attributes.

`shutil.`**`copy`**(*src*, *dst*, **, *follow_symlinks=True*)

Copies the file *src* to the file or directory *dst*. *src* and *dst* should be *path-like objects* or strings. If *dst* specifies a directory, the file will be copied into *dst* using the base filename from *src*. If *dst* specifies a file that already exists, it will be replaced. Returns the path to the newly created file.

If *follow_symlinks* is false, and *src* is a symbolic link, *dst* will be created as a symbolic link. If *follow_symlinks* is true and *src* is a symbolic link, *dst* will be a copy of the file *src* refers to.

`copy()` copies the file data and the file's permission mode (see `os.chmod()`). Other metadata, like the file's creation and modification times, is not preserved. To preserve all file metadata from the original, use `copy2()` instead.

Lève un *événement d'audit* `shutil.copyfile` avec les arguments `src`, `dst`.

Lève un *événement d'audit* `shutil.copymode` avec les arguments `src`, `dst`.

Modifié dans la version 3.3 : Added *follow_symlinks* argument. Now returns path to the newly created file.

Modifié dans la version 3.8 : Les appels système de copie rapide spécifiques à la plate-forme peuvent être utilisés en interne afin de copier le fichier plus efficacement. Voir la section *Platform-dependent efficient copy operations*.

`shutil.`**`copy2`**(*src*, *dst*, **, *follow_symlinks=True*)

Identical to `copy()` except that `copy2()` also attempts to preserve file metadata.

When *follow_symlinks* is false, and *src* is a symbolic link, `copy2()` attempts to copy all metadata from the *src* symbolic link to the newly created *dst* symbolic link. However, this functionality is not available on all platforms. On platforms where some or all of this functionality is unavailable, `copy2()` will preserve all the metadata it can ; `copy2()` never raises an exception because it cannot preserve file metadata.

`copy2()` uses `copystat()` to copy the file metadata. Please see `copystat()` for more information about platform support for modifying symbolic link metadata.

Lève un *événement d'audit* `shutil.copyfile` avec les arguments `src`, `dst`.

Raises an *auditing event* `shutil.copystat` with arguments `src`, `dst`.

Modifié dans la version 3.3 : Added *follow_symlinks* argument, try to copy extended file system attributes too (currently Linux only). Now returns path to the newly created file.

Modifié dans la version 3.8 : Les appels système de copie rapide spécifiques à la plate-forme peuvent être utilisés en interne afin de copier le fichier plus efficacement. Voir la section *Platform-dependent efficient copy operations*.

`shutil.`**`ignore_patterns`**(**patterns*)

This factory function creates a function that can be used as a callable for `copytree()`'s *ignore* argument, ignoring files and directories that match one of the glob-style *patterns* provided. See the example below.

`shutil.`**`copytree`**(*src*, *dst*, *symlinks=False*, *ignore=None*, *copy_function=copy2*, *ignore_dangling_symlinks=False*, *dirs_exist_ok=False*)

Recursively copy an entire directory tree rooted at *src* to a directory named *dst* and return the destination directory. All intermediate directories needed to contain *dst* will also be created by default.

Permissions and times of directories are copied with `copystat()`, individual files are copied using `copy2()`.

If *symlinks* is true, symbolic links in the source tree are represented as symbolic links in the new tree and the metadata of the original links will be copied as far as the platform allows ; if false or omitted, the contents and metadata of the linked files are copied to the new tree.

When *symlinks* is false, if the file pointed to by the symlink doesn't exist, an exception will be added in the list of errors raised in an `Error` exception at the end of the copy process. You can set the optional *ignore_dangling_symlinks* flag to true if you want to silence this exception. Notice that this option has no effect on platforms that don't support `os.symlink()`.

If *ignore* is given, it must be a callable that will receive as its arguments the directory being visited by `copytree()`, and a list of its contents, as returned by `os.listdir()`. Since `copytree()` is called recursively, the *ignore* callable will be called once for each directory that is copied. The callable must return a sequence of directory and file names relative to the current directory (i.e. a subset of the items in its second argument) ; these names will then be ignored in the copy process. `ignore_patterns()` can be used to create such a callable that ignores names based on glob-style patterns.

If exception(s) occur, an `Error` is raised with a list of reasons.

If *copy_function* is given, it must be a callable that will be used to copy each file. It will be called with the source path and the destination path as arguments. By default, `copy2()` is used, but any function that supports the same signature (like `copy()`) can be used.

If *dirs_exist_ok* is false (the default) and *dst* already exists, a `FileExistsError` is raised. If *dirs_exist_ok* is true, the copying operation will continue if it encounters existing directories, and files within the *dst* tree will be overwritten by corresponding files from the *src* tree.

Raises an *auditing event* `shutil.copytree` with arguments `src`, `dst`.

Modifié dans la version 3.2 : Added the *copy_function* argument to be able to provide a custom copy function. Added the *ignore_dangling_symlinks* argument to silence dangling symlinks errors when *symlinks* is false.

Modifié dans la version 3.3 : Copy metadata when *symlinks* is false. Now returns *dst*.

Modifié dans la version 3.8 : Les appels système de copie rapide spécifiques à la plate-forme peuvent être utilisés en interne afin de copier le fichier plus efficacement. Voir la section *Platform-dependent efficient copy operations*.

Modifié dans la version 3.8 : Added the *dirs_exist_ok* parameter.

`shutil.`**`rmtree`**(*path*, *ignore_errors=False*, *onerror=None*, *, *onexc=None*, *dir_fd=None*)

Delete an entire directory tree ; *path* must point to a directory (but not a symbolic link to a directory). If *ignore_errors* is true, errors resulting from failed removals will be ignored ; if false or omitted, such errors are handled by calling a handler specified by *onexc* or *onerror* or, if both are omitted, exceptions are propagated to the caller.

Cette fonction prend en charge *des chemins relatifs à des descripteurs de répertories*.

> **Note**
>
> On platforms that support the necessary fd-based functions a symlink attack resistant version of `rmtree()` is used by default. On other platforms, the `rmtree()` implementation is susceptible to a symlink attack : given proper timing and circumstances, attackers can manipulate symlinks on the filesystem to delete files they wouldn't be able to access otherwise. Applications can use the `rmtree.avoids_symlink_attacks` function attribute to determine which case applies.

If *onexc* is provided, it must be a callable that accepts three parameters : *function*, *path*, and *excinfo*.

The first parameter, *function*, is the function which raised the exception ; it depends on the platform and implementation. The second parameter, *path*, will be the path name passed to *function*. The third parameter, *excinfo*, is the exception that was raised. Exceptions raised by *onexc* will not be caught.

The deprecated *onerror* is similar to *onexc*, except that the third parameter it receives is the tuple returned from `sys.exc_info()`.

Raises an *auditing event* `shutil.rmtree` with arguments `path`, `dir_fd`.

Modifié dans la version 3.3 : Added a symlink attack resistant version that is used automatically if platform supports fd-based functions.

Modifié dans la version 3.8 : On Windows, will no longer delete the contents of a directory junction before removing the junction.

Modifié dans la version 3.11 : Added the *dir_fd* parameter.

Modifié dans la version 3.12 : Added the *onexc* parameter, deprecated *onerror*.

Modifié dans la version 3.13 : `rmtree()` now ignores `FileNotFoundError` exceptions for all but the top-level path. Exceptions other than `OSError` and subclasses of `OSError` are now always propagated to the caller.

`rmtree.`**`avoids_symlink_attacks`**

Indicates whether the current platform and implementation provides a symlink attack resistant version of `rmtree()`. Currently this is only true for platforms supporting fd-based directory access functions.

Ajouté dans la version 3.3.

`shutil.move`(*src*, *dst*, *copy_function=copy2*)

Recursively move a file or directory (*src*) to another location and return the destination.

If *dst* is an existing directory or a symlink to a directory, then *src* is moved inside that directory. The destination path in that directory must not already exist.

If *dst* already exists but is not a directory, it may be overwritten depending on `os.rename()` semantics.

If the destination is on the current filesystem, then `os.rename()` is used. Otherwise, *src* is copied to the destination using *copy_function* and then removed. In case of symlinks, a new symlink pointing to the target of *src* will be created as the destination and *src* will be removed.

If *copy_function* is given, it must be a callable that takes two arguments, *src* and the destination, and will be used to copy *src* to the destination if `os.rename()` cannot be used. If the source is a directory, `copytree()` is called, passing it the *copy_function*. The default *copy_function* is `copy2()`. Using `copy()` as the *copy_function* allows the move to succeed when it is not possible to also copy the metadata, at the expense of not copying any of the metadata.

Raises an *auditing event* `shutil.move` with arguments `src`, `dst`.

Modifié dans la version 3.3 : Added explicit symlink handling for foreign filesystems, thus adapting it to the behavior of GNU's **mv**. Now returns *dst*.

Modifié dans la version 3.5 : Added the *copy_function* keyword argument.

Modifié dans la version 3.8 : Les appels système de copie rapide spécifiques à la plate-forme peuvent être utilisés en interne afin de copier le fichier plus efficacement. Voir la section *Platform-dependent efficient copy operations*.

Modifié dans la version 3.9 : Accepts a *path-like object* for both *src* and *dst*.

`shutil.disk_usage`(*path*)

Return disk usage statistics about the given path as a *named tuple* with the attributes *total*, *used* and *free*, which are the amount of total, used and free space, in bytes. *path* may be a file or a directory.

> **Note**
>
> On Unix filesystems, *path* must point to a path within a **mounted** filesystem partition. On those platforms, CPython doesn't attempt to retrieve disk usage information from non-mounted filesystems.

Ajouté dans la version 3.3.

Modifié dans la version 3.8 : On Windows, *path* can now be a file or directory.

Availability : Unix, Windows.

`shutil.chown`(*path*, *user=None*, *group=None*, ***, *dir_fd=None*, *follow_symlinks=True*)

Change owner *user* and/or *group* of the given *path*.

user can be a system user name or a uid ; the same applies to *group*. At least one argument is required.

See also `os.chown()`, the underlying function.

Raises an *auditing event* `shutil.chown` with arguments `path`, `user`, `group`.

Availability : Unix.

Ajouté dans la version 3.3.

Modifié dans la version 3.13 : Added *dir_fd* and *follow_symlinks* parameters.

`shutil.which`(*cmd*, *mode=os.F_OK | os.X_OK*, *path=None*)

Return the path to an executable which would be run if the given *cmd* was called. If no *cmd* would be called, return `None`.

mode is a permission mask passed to `os.access()`, by default determining if the file exists and is executable.

path is a "`PATH` string" specifying the directories to look in, delimited by `os.pathsep`. When no *path* is specified, the `PATH` environment variable is read from `os.environ`, falling back to `os.defpath` if it is not set.

On Windows, the current directory is prepended to the *path* if *mode* does not include `os.X_OK`. When the *mode* does include `os.X_OK`, the Windows API `NeedCurrentDirectoryForExePathW` will be consulted to determine if the current directory should be prepended to *path*. To avoid consulting the current working directory for executables : set the environment variable `NoDefaultCurrentDirectoryInExePath`.

Also on Windows, the `PATHEXT` environment variable is used to resolve commands that may not already include an extension. For example, if you call `shutil.which("python")`, `which()` will search `PATHEXT` to know that it should look for `python.exe` within the *path* directories. For example, on Windows :

```
>>> shutil.which("python")
'C:\\Python33\\python.EXE'
```

This is also applied when *cmd* is a path that contains a directory component :

```
>> shutil.which("C:\\Python33\\python")
'C:\\Python33\\python.EXE'
```

Ajouté dans la version 3.3.

Modifié dans la version 3.8 : The `bytes` type is now accepted. If *cmd* type is `bytes`, the result type is also `bytes`.

Modifié dans la version 3.12 : On Windows, the current directory is no longer prepended to the search path if *mode* includes `os.X_OK` and WinAPI `NeedCurrentDirectoryForExePathW(cmd)` is false, else the current directory is prepended even if it is already in the search path; `PATHEXT` is used now even when *cmd* includes a directory component or ends with an extension that is in `PATHEXT`; and filenames that have no extension can now be found.

exception `shutil.`**`Error`**

This exception collects exceptions that are raised during a multi-file operation. For `copytree()`, the exception argument is a list of 3-tuples (*srcname*, *dstname*, *exception*).

Platform-dependent efficient copy operations

Starting from Python 3.8, all functions involving a file copy (`copyfile()`, `copy()`, `copy2()`, `copytree()`, and `move()`) may use platform-specific "fast-copy" syscalls in order to copy the file more efficiently (see bpo-33671). "fast-copy" means that the copying operation occurs within the kernel, avoiding the use of userspace buffers in Python as in "`outfd.write(infd.read())`".

On macOS fcopyfile is used to copy the file content (not metadata).

On Linux `os.copy_file_range()` or `os.sendfile()` is used.

On Solaris `os.sendfile()` is used.

On Windows `shutil.copyfile()` uses a bigger default buffer size (1 MiB instead of 64 KiB) and a `memoryview()`-based variant of `shutil.copyfileobj()` is used.

If the fast-copy operation fails and no data was written in the destination file then shutil will silently fallback on using less efficient `copyfileobj()` function internally.

Modifié dans la version 3.8.

Modifié dans la version 3.14 : Solaris now uses `os.sendfile()`.

Modifié dans la version 3.14 : Copy-on-write or server-side copy may be used internally via `os.copy_file_range()` on supported Linux filesystems.

copytree example

An example that uses the `ignore_patterns()` helper :

```
from shutil import copytree, ignore_patterns

copytree(source, destination, ignore=ignore_patterns('*.pyc', 'tmp*'))
```

This will copy everything except `.pyc` files and files or directories whose name starts with `tmp`.

Another example that uses the *ignore* argument to add a logging call :

```
from shutil import copytree
import logging

def _logpath(path, names):
    logging.info('Working in %s', path)
    return []    # nothing will be ignored

copytree(source, destination, ignore=_logpath)
```

rmtree example

This example shows how to remove a directory tree on Windows where some of the files have their read-only bit set. It uses the onexc callback to clear the readonly bit and reattempt the remove. Any subsequent failure will propagate.

```
import os, stat
import shutil

def remove_readonly(func, path, _):
    "Clear the readonly bit and reattempt the removal"
    os.chmod(path, stat.S_IWRITE)
    func(path)

shutil.rmtree(directory, onexc=remove_readonly)
```

11.9.2 Archiving operations

Ajouté dans la version 3.2.

Modifié dans la version 3.5 : Added support for the *xztar* format.

High-level utilities to create and read compressed and archived files are also provided. They rely on the `zipfile` and `tarfile` modules.

`shutil.`**`make_archive`**(*base_name*, *format*[, *root_dir*[, *base_dir*[, *verbose*[, *dry_run*[, *owner*[, *group*[, *logger*]]]]]]])

Create an archive file (such as zip or tar) and return its name.

base_name is the name of the file to create, including the path, minus any format-specific extension.

format is the archive format : one of "zip" (if the `zlib` module is available), "tar", "gztar" (if the `zlib` module is available), "bztar" (if the `bz2` module is available), or "xztar" (if the `lzma` module is available).

root_dir is a directory that will be the root directory of the archive, all paths in the archive will be relative to it ; for example, we typically chdir into *root_dir* before creating the archive.

base_dir is the directory where we start archiving from ; i.e. *base_dir* will be the common prefix of all files and directories in the archive. *base_dir* must be given relative to *root_dir*. See *Archiving example with base_dir* for how to use *base_dir* and *root_dir* together.

root_dir and *base_dir* both default to the current directory.

If *dry_run* is true, no archive is created, but the operations that would be executed are logged to *logger*.

owner and *group* are used when creating a tar archive. By default, uses the current owner and group.

logger must be an object compatible with **PEP 282**, usually an instance of `logging.Logger`.

The *verbose* argument is unused and deprecated.

Raises an *auditing event* `shutil.make_archive` with arguments `base_name`, `format`, `root_dir`, `base_dir`.

> **Note**
>
> This function is not thread-safe when custom archivers registered with `register_archive_format()` do not support the *root_dir* argument. In this case it temporarily changes the current working directory of the process to *root_dir* to perform archiving.

Modifié dans la version 3.8 : The modern pax (POSIX.1-2001) format is now used instead of the legacy GNU format for archives created with `format="tar"`.

Modifié dans la version 3.10.6 : This function is now made thread-safe during creation of standard `.zip` and tar archives.

`shutil.`**`get_archive_formats`**`()`

Return a list of supported formats for archiving. Each element of the returned sequence is a tuple `(name, description)`.

By default `shutil` provides these formats :
— *zip* : ZIP file (if the `zlib` module is available).
— *tar* : Uncompressed tar file. Uses POSIX.1-2001 pax format for new archives.
— *gztar* : gzip'ed tar-file (if the `zlib` module is available).
— *bztar* : bzip2'ed tar-file (if the `bz2` module is available).
— *xztar* : xz'ed tar-file (if the `lzma` module is available).
You can register new formats or provide your own archiver for any existing formats, by using `register_archive_format()`.

`shutil.`**`register_archive_format`**`(`*name*, *function*[, *extra_args*[, *description*]]`)`

Register an archiver for the format *name*.

function is the callable that will be used to unpack archives. The callable will receive the *base_name* of the file to create, followed by the *base_dir* (which defaults to `os.curdir`) to start archiving from. Further arguments are passed as keyword arguments : *owner*, *group*, *dry_run* and *logger* (as passed in `make_archive()`).

If *function* has the custom attribute `function.supports_root_dir` set to `True`, the *root_dir* argument is passed as a keyword argument. Otherwise the current working directory of the process is temporarily changed to *root_dir* before calling *function*. In this case `make_archive()` is not thread-safe.

If given, *extra_args* is a sequence of `(name, value)` pairs that will be used as extra keywords arguments when the archiver callable is used.

description is used by `get_archive_formats()` which returns the list of archivers. Defaults to an empty string.

Modifié dans la version 3.12 : Added support for functions supporting the *root_dir* argument.

`shutil.`**`unregister_archive_format`**`(`*name*`)`

Remove the archive format *name* from the list of supported formats.

`shutil.`**`unpack_archive`**`(`*filename*[, *extract_dir*[, *format*[, *filter*]]]`)`

Unpack an archive. *filename* is the full path of the archive.

extract_dir is the name of the target directory where the archive is unpacked. If not provided, the current working directory is used.

format is the archive format : one of "zip", "tar", "gztar", "bztar", or "xztar". Or any other format registered

with `register_unpack_format()`. If not provided, `unpack_archive()` will use the archive file name extension and see if an unpacker was registered for that extension. In case none is found, a `ValueError` is raised.

The keyword-only *filter* argument is passed to the underlying unpacking function. For zip files, *filter* is not accepted. For tar files, it is recommended to use `'data'` (default since Python 3.14), unless using features specific to tar and UNIX-like filesystems. (See *Extraction filters* for details.)

Raises an *auditing event* `shutil.unpack_archive` with arguments `filename`, `extract_dir`, `format`.

> **Avertissement**
>
> Never extract archives from untrusted sources without prior inspection. It is possible that files are created outside of the path specified in the *extract_dir* argument, e.g. members that have absolute filenames starting with "/" or filenames with two dots "..".
>
> Since Python 3.14, the defaults for both built-in formats (zip and tar files) will prevent the most dangerous of such security issues, but will not prevent *all* unintended behavior. Read the *Hints for further verification* section for tar-specific details.

Modifié dans la version 3.7 : Accepts a *path-like object* for *filename* and *extract_dir*.

Modifié dans la version 3.12 : Added the *filter* argument.

`shutil.`**`register_unpack_format`**(*name*, *extensions*, *function*[, *extra_args*[, *description*]])

Registers an unpack format. *name* is the name of the format and *extensions* is a list of extensions corresponding to the format, like `.zip` for Zip files.

function is the callable that will be used to unpack archives. The callable will receive :
— the path of the archive, as a positional argument ;
— the directory the archive must be extracted to, as a positional argument ;
— possibly a *filter* keyword argument, if it was given to `unpack_archive()` ;
— additional keyword arguments, specified by *extra_args* as a sequence of `(name, value)` tuples.
description can be provided to describe the format, and will be returned by the `get_unpack_formats()` function.

`shutil.`**`unregister_unpack_format`**(*name*)

Unregister an unpack format. *name* is the name of the format.

`shutil.`**`get_unpack_formats`**()

Return a list of all registered formats for unpacking. Each element of the returned sequence is a tuple `(name, extensions, description)`.

By default `shutil` provides these formats :
— *zip* : ZIP file (unpacking compressed files works only if the corresponding module is available).
— *tar* : uncompressed tar file.
— *gztar* : gzip'ed tar-file (if the `zlib` module is available).
— *bztar* : bzip2'ed tar-file (if the `bz2` module is available).
— *xztar* : xz'ed tar-file (if the `lzma` module is available).
You can register new formats or provide your own unpacker for any existing formats, by using `register_unpack_format()`.

Archiving example

In this example, we create a gzip'ed tar-file archive containing all files found in the `.ssh` directory of the user :

```
>>> from shutil import make_archive
>>> import os
>>> archive_name = os.path.expanduser(os.path.join('~', 'myarchive'))
>>> root_dir = os.path.expanduser(os.path.join('~', '.ssh'))
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> make_archive(archive_name, 'gztar', root_dir)
'/Users/tarek/myarchive.tar.gz'
```

The resulting archive contains :

```
$ tar -tzvf /Users/tarek/myarchive.tar.gz
drwx------ tarek/staff       0 2010-02-01 16:23:40 ./
-rw-r--r-- tarek/staff     609 2008-06-09 13:26:54 ./authorized_keys
-rwxr-xr-x tarek/staff      65 2008-06-09 13:26:54 ./config
-rwx------ tarek/staff     668 2008-06-09 13:26:54 ./id_dsa
-rwxr-xr-x tarek/staff     609 2008-06-09 13:26:54 ./id_dsa.pub
-rw------- tarek/staff    1675 2008-06-09 13:26:54 ./id_rsa
-rw-r--r-- tarek/staff     397 2008-06-09 13:26:54 ./id_rsa.pub
-rw-r--r-- tarek/staff   37192 2010-02-06 18:23:10 ./known_hosts
```

Archiving example with *base_dir*

In this example, similar to the *one above*, we show how to use `make_archive()`, but this time with the usage of *base_dir*. We now have the following directory structure :

```
$ tree tmp
tmp
└── root
    └── structure
        ├── content
            └── please_add.txt
        └── do_not_add.txt
```

In the final archive, `please_add.txt` should be included, but `do_not_add.txt` should not. Therefore we use the following :

```
>>> from shutil import make_archive
>>> import os
>>> archive_name = os.path.expanduser(os.path.join('~', 'myarchive'))
>>> make_archive(
...     archive_name,
...     'tar',
...     root_dir='tmp/root',
...     base_dir='structure/content',
... )
'/Users/tarek/my_archive.tar'
```

Listing the files in the resulting archive gives us :

```
$ python -m tarfile -l /Users/tarek/myarchive.tar
structure/content/
structure/content/please_add.txt
```

11.9.3 Querying the size of the output terminal

`shutil.get_terminal_size`(*fallback=(columns, lines)*)

Get the size of the terminal window.

For each of the two dimensions, the environment variable, `COLUMNS` and `LINES` respectively, is checked. If the variable is defined and the value is a positive integer, it is used.

When `COLUMNS` or `LINES` is not defined, which is the common case, the terminal connected to `sys.__stdout__` is queried by invoking `os.get_terminal_size()`.

If the terminal size cannot be successfully queried, either because the system doesn't support querying, or because we are not connected to a terminal, the value given in `fallback` parameter is used. `fallback` defaults to `(80, 24)` which is the default size used by many terminal emulators.

The value returned is a named tuple of type `os.terminal_size`.

See also : The Single UNIX Specification, Version 2, Other Environment Variables.

Ajouté dans la version 3.3.

Modifié dans la version 3.11 : The `fallback` values are also used if `os.get_terminal_size()` returns zeroes.

Voir aussi

Module `os`
Interfaces du système d'exploitation, incluant des fonctions pour travailler avec des fichiers dans un niveau plus bas que les *objets fichiers* de Python.

Module `io`
Bibliothèque d'entrées/sorties native de Python, incluant des classes abstraites et concrètes tel que les I/O sur les fichiers.

Fonction native `open()`
Le moyen classique pour ouvrir des fichiers pour les lire ou y écrire avec Python.

CHAPITRE 12

Persistance des données

Les modules décrits dans ce chapitre permettent de stocker des données Python de manière persistante typiquement sur disque. Les modules `pickle` et `marshal` peuvent transformer n'importe quel type Python en une séquence d'octets, puis recréer les objets depuis ces octets. Les différents modules du paquet *dbm* gèrent une catégorie de formats de fichier basée sur des hach, stockant des correspondances entre chaînes de caractères.

La liste des modules documentés dans ce chapitre est :

12.1 `pickle` --- Python object serialization

Code source : Lib/pickle.py

Le module `pickle` implémente des protocoles binaires de sérialisation et dé-sérialisation d'objets Python. La sérialisation est le procédé par lequel une hiérarchie d'objets Python est convertie en flux d'octets. La désérialisation est l'opération inverse, par laquelle un flux d'octets (à partir d'un *binary file* ou *bytes-like object*) est converti en hiérarchie d'objets. Sérialisation (et *désérialisation*) sont aussi connus sous les termes de *pickling*, de "*marshalling*" [1] ou encore de "*flattening*".

Avertissement

Le module `pickle` **n'est pas sécurisé**. Ne désérialisez des objets qu'à partir de sources fiables.

Il est possible de produire des données binaires qui **exécutent du code arbitraire lors de leur désérialisation**. Ne désérialisez jamais des données provenant d'une source non fiable, ou qui pourraient avoir été modifiées.

Pensez au module `hmac` pour signer des données afin de s'assurer qu'elles n'ont pas été modifiées.

Des formats de sérialisation plus sûrs, comme `json`, peuvent se révéler plus adaptés si vous travaillez sur des données qui ne sont pas fiables. Voir *Comparaison avec json*.

12.1.1 Relations aux autres modules Python

Comparaison avec `marshal`

Python possède un module de bas niveau en sérialisation appelé `marshal`, mais en général il est préférable d'utiliser `pickle` pour sérialiser des objets Python. `marshal` existe principalement pour gérer les fichiers Python en `.pyc`.

1. À ne pas confondre avec ce que fait le module `marshal`.

Le module `pickle` diffère du module `marshal` sur plusieurs aspects :

- Le module `pickle` garde la trace des objets qu'il a déjà sérialisés, pour faire en sorte que les prochaines références à cet objet ne soient pas sérialisées à nouveau. `marshal` ne le fait pas.

  Ça a des implications sur les objets partagés et les objets récursifs. Les objets récursifs sont des objets qui contiennent des références à eux-mêmes. Ceux-ci ne sont pas gérées par marshal : lui donner un objet récursif va le faire planter. Un objet est partagé lorsque que plusieurs références pointent dessus, depuis différents endroits dans la hiérarchie sérialisée. Le module `pickle` repère ces partages et ne stocke ces objets qu'une seule fois. Les objets partagés restent ainsi partagés, ce qui peut être très important pour les objets mutables.
- `marshal` ne peut être utilisé pour la sérialisation et l'instanciation de classes définies par les utilisateurs. `pickle` peut sauvegarder et restaurer les instances de classes de manière transparente. Cependant la définition de classe doit être importable et lancée dans le même module et de la même manière que lors de son importation.
- Aucune garantie n'est offerte sur la portabilité du format `marshal` entre différentes versions de Python. Sa fonction première étant la gestion des fichiers `.pyc`, les développeurs se réservent le droit de changer le format de sérialisation de manière non-rétrocompatible si besoin était. Il est garanti que le format `pickle` restera compatible avec les versions futures de Python, pourvu que vous choisissiez un protocole de sérialisation adapté. De plus, il masque les différences entre les types Python 2 et Python 3, pour le cas où il s'agit de désérialiser en Python 3 des données sérialisées en Python 2.

Comparaison avec `json`

There are fundamental differences between the pickle protocols and JSON (JavaScript Object Notation) :

- *pickle* est un format binaire, tandis que JSON est un format textuel (constitué de caractères Unicode et généralement encodé en UTF-8) ;
- JSON peut être lu par une personne, contrairement à *pickle* ;
- JSON offre l'interopérabilité avec de nombreux outils en dehors de l'écosystème Python, alors que *pickle* est propre à Python ;
- Par défaut, JSON n'est capable de sérialiser qu'un nombre limité de types natifs Python, et ne prend pas en charge les classes définies par l'utilisateur. Le format *pickle* peut représenter une multitude de types d'objets, dont beaucoup automatiquement, grâce à une utilisation fine des possibilités d'introspection de Python ; on peut traiter les cas les plus complexes en implémentant des *méthodes de sérialisation propres à une classe* ;
- Contrairement à *pickle*, la désérialisation de données JSON n'ouvre pas en soi une vulnérabilité à l'exécution de code arbitraire.

Voir aussi

Le module `json` de la bibliothèque standard permet la sérialisation et désérialisation au format JSON.

12.1.2 Format du flux de données

The data format used by `pickle` is Python-specific. This has the advantage that there are no restrictions imposed by external standards such as JSON (which can't represent pointer sharing) ; however it means that non-Python programs may not be able to reconstruct pickled Python objects.

Le format binaire `pickle` est, par défaut, une représentation assez compacte des objets. Il est possible de *compresser* efficacement les données sérialisées.

Le module `pickletools` contient des outils servant à analyser les flux de données générés par `pickle`. Le code source de `pickletools` contient des commentaires détaillés sur les *opcodes* employés par les protocoles *pickle*.

Il existe actuellement 6 protocoles différents pour la sérialisation. Les protocoles portant les numéros les plus grands sont les derniers ajoutés, et nécessitent en conséquence des versions de Python plus récentes.

- Le protocole 0 est le format originel, humainement lisible. Il est rétrocompatible avec les versions les plus anciennes de Python.
- Le protocole 1 est un ancien format binaire, aussi compatible avec les versions anciennes.
- Protocol version 2 was introduced in Python 2.3. It provides much more efficient pickling of *new-style classes*. Refer to **PEP 307** for information about improvements brought by protocol 2.
- Le protocole 3 a été introduit en Python 3.0. Il gère les objets `bytes`, et ne permet pas la désérialisation par Python 2.x. Il fut le protocole par défaut de Python 3.0 à Python 3.7.

— Protocol version 4 was added in Python 3.4. It adds support for very large objects, pickling more kinds of objects, and some data format optimizations. This was the default protocol in Python 3.8--3.13. Refer to **PEP 3154** for information about improvements brought by protocol 4.
— Protocol version 5 was added in Python 3.8. It adds support for out-of-band data and speedup for in-band data. It is the default protocol starting with Python 3.14. Refer to **PEP 574** for information about improvements brought by protocol 5.

Note

La sérialisation est un problème plus simple que la persistance des données en général. Le module `pickle` lit et écrit des objets fichiers-compatibles, mais ne s'occupe pas du problème de donner un nom à des objets persistants, ni de gérer l'accès par différents processus en parallèle à ces objets. `pickle` se contente de transformer des objets complexes en flux d'octets, et lire ces flux d'octets par la suite pour reconstruire des objets avec la même structure. Si l'on peut bien sûr écrire les flux d'octets dans un fichier, rien n'empêche de les transférer à travers un réseau, ou bien de les stocker dans une base de données. Voir le module `shelve` pour une interface simple qui sérialise et désérialise les objets dans des bases de données de style DBM.

12.1.3 Interface du module

Pour sérialiser un objet, contenant éventuellement d'autres objets, appelez tout simplement la fonction `dumps()`. La fonction `loads()`, quant à elle, désérialise un flux de données. Pour un contrôle plus fin des opérations de sérialisation ou désérialisation, créez un objet `Pickler` ou `Unpickler`.

Le module `pickle` définit les constantes suivantes :

`pickle.`**`HIGHEST_PROTOCOL`**

Entier qui donne la version du *protocole* le plus récent qui soit disponible. Ce nombre peut être passé comme paramètre *protocol* aux fonctions `dump()` et `dumps()` ainsi qu'au constructeur de la classe `Pickler`.

`pickle.`**`DEFAULT_PROTOCOL`**

An integer, the default *protocol version* used for pickling. May be less than `HIGHEST_PROTOCOL`. Currently the default protocol is 5, introduced in Python 3.8 and incompatible with previous versions. This version introduces support for out-of-band buffers, where **PEP 3118**-compatible data can be transmitted separately from the main pickle stream.

Modifié dans la version 3.0 : Le protocole par défaut est devenu le protocole 3.

Modifié dans la version 3.8 : Le protocole par défaut est devenu le protocole 4.

Modifié dans la version 3.14 : The default protocol is 5.

Le module `pickle` contient quelques fonctions pour faciliter la sérialisation :

`pickle.`**`dump`**(*obj*, *file*, *protocol=None*, **, fix_imports=True*, *buffer_callback=None*)

Écrit la représentation sérialisée de l'objet *obj* dans l'*objet fichier-compatible file*, qui doit être ouvert. Ceci est l'équivalent de `Pickle(file, protocol).dump(obj)`.

Les arguments *file*, *protocol*, *fix_imports* et *buffer_callback* sont identiques à ceux du constructeur de la classe `Pickler`.

Modifié dans la version 3.8 : ajout de l'argument *buffer_callback*.

`pickle.`**`dumps`**(*obj*, *protocol=None*, **, fix_imports=True*, *buffer_callback=None*)

Renvoie la représentation sérialisée de *obj* sous forme de `bytes`, au lieu de l'écrire dans un fichier.

Les arguments *protocol*, *fix_imports* et *buffer_callback* sont identiques à ceux du constructeur de la classe `Pickler`.

Modifié dans la version 3.8 : ajout de l'argument *buffer_callback*.

`pickle.`**`load`**(*file*, **, fix_imports=True*, *encoding='ASCII'*, *errors='strict'*, *buffers=None*)

Charge la représentation sérialisée d'un objet depuis l'*objet fichier-compatible* ouvert *file*, et renvoie l'objet reconstitué obtenu. Ceci est l'équivalent de `Unpickler(file).load()`.

La version du protocole utilisée pour la sérialisation est détectée automatiquement, d'où l'absence d'un argument. Les octets situés après la représentation sérialisée de l'objet sont ignorés.

Les arguments *file*, *fix_imports*, *encoding*, *errors*, *strict* et *buffers* sont identiques à ceux du constructeur de la classe `Unpickler`.

Modifié dans la version 3.8 : Ajout de l'argument *buffers*.

`pickle.`**`loads`**(*data*, /, *, *fix_imports=True*, *encoding='ASCII'*, *errors='strict'*, *buffers=None*)

Renvoie l'objet reconstitué à partir de la représentation sérialisée *data*, qui doit être fournie sous la forme d'un *bytes-like object*.

La version du protocole utilisée pour la sérialisation est détectée automatiquement, d'où l'absence d'un argument. Les octets situés après la représentation sérialisée de l'objet sont ignorés.

Arguments *fix_imports*, *encoding*, *errors*, *strict* and *buffers* have the same meaning as in the `Unpickler` constructor.

Modifié dans la version 3.8 : Ajout de l'argument *buffers*.

Le module `pickle` définit trois types d'exceptions :

exception `pickle.`**`PickleError`**

Common base class for the other pickling exceptions. It inherits from `Exception`.

exception `pickle.`**`PicklingError`**

Error raised when an unpicklable object is encountered by `Pickler`. It inherits from `PickleError`.

Lisez *Quels objets sont sérialisables ?* pour en savoir plus sur les types d'objets qui peuvent être sérialisés.

exception `pickle.`**`UnpicklingError`**

Error raised when there is a problem unpickling an object, such as a data corruption or a security violation. It inherits from `PickleError`.

Veuillez noter que d'autres exceptions peuvent être levées durant la désérialisation, comme `AttributeError`, `EOFError`, `ImportError` et `IndexError` (liste non-exhaustive).

Le module `pickle` exporte trois classes : `Pickler`, `Unpickler` et `PickleBuffer` :

class `pickle.`**`Pickler`**(*file*, *protocol=None*, *, *fix_imports=True*, *buffer_callback=None*)

Classe d'objets qui implémentent la sérialisation vers un flux binaire.

L'argument optionnel *protocol* détermine la version du protocole de sérialisation à employer, entre 0 et `HIGHEST_PROTOCOL`. La valeur par défaut est celle de `DEFAULT_PROTOCOL`. Avec un nombre strictement négatif, c'est `HIGHEST_PROTOCOL` qui est utilisé.

L'argument *file* peut être un fichier sur disque ouvert en mode binaire pour l'écriture, une instance de la classe `io.BytesIO`, ou plus généralement un objet quelconque qui possède une méthode `write()` acceptant d'être appelée sur un argument unique de type `bytes`.

Si *fix_imports* est vrai et *protocol* est inférieur ou égal à 2, les noms des modules sont reliés par *pickle* aux anciens noms qui avaient cours en Python 2, afin que le flux sérialisé soit lisible aussi bien par Python 2 que Python 3.

If *buffer_callback* is `None` (the default), buffer views are serialized into *file* as part of the pickle stream.

If *buffer_callback* is not `None`, then it can be called any number of times with a buffer view. If the callback returns a false value (such as `None`), the given buffer is *out-of-band* ; otherwise the buffer is serialized in-band, i.e. inside the pickle stream.

It is an error if *buffer_callback* is not `None` and *protocol* is `None` or smaller than 5.

Modifié dans la version 3.8 : ajout de l'argument *buffer_callback*.

dump(*obj*)

Écrit la représentation sérialisée de l'objet *obj* dans le fichier ouvert passé au constructeur.

persistent_id(*obj*)

Ne fait rien par défaut. Cette méthode est destinée à être implémentée par une classe fille.

Si `persistent_id()` renvoie `None`, *obj* est sérialisé normalement. Toute autre valeur est reprise par le `Pickler` comme ID persistant pour *obj*. Le sens de cet ID persistant doit être défini par `Unpickler.persistent_load()`. Veuillez noter que la valeur renvoyée par `persistent_id()` ne peut pas porter elle-même d'ID persistant.

La section *Persistance d'objets externes* donne des détails et exemples.

Modifié dans la version 3.13 : Add the default implementation of this method in the C implementation of `Pickler`.

dispatch_table

A pickler object's dispatch table is a registry of *reduction functions* of the kind which can be declared using `copyreg.pickle()`. It is a mapping whose keys are classes and whose values are reduction functions. A reduction function takes a single argument of the associated class and should conform to the same interface as a `__reduce__()` method.

Lorsqu'un sérialiseur ne possède pas l'attribut `dispatch_table`, comme c'est le cas par défaut, il utilise le tableau global du module `copyreg`. Afin de personnaliser l'opération de sérialisation pour un sérialiseur particulier, on peut affecter à son attribut `dispatch_table` un objet compatible avec les dictionnaires. Une autre possibilité est de définir l'attribut `dispatch_table` dans une classe fille de `Pickler`. Sa valeur sera alors utilisée pour toutes les instances de cette classe fille.

Voir *Tables de distribution* pour des exemples d'utilisation.

Ajouté dans la version 3.3.

reducer_override(*obj*)

Special reducer that can be defined in `Pickler` subclasses. This method has priority over any reducer in the `dispatch_table`. It should conform to the same interface as a `__reduce__()` method, and can optionally return `NotImplemented` to fallback on `dispatch_table`-registered reducers to pickle `obj`.

Voir *Réduction personnalisée pour les types, fonctions et autres objets* pour un exemple détaillé.

Ajouté dans la version 3.8.

fast

Cet attribut est obsolète. Une valeur vraie (« mode rapide ») désactive la mémorisation des objets au fur et à mesure de leur sérialisation, qui permet habituellement de représenter les doublons par une unique référence. Cette option accélère la sérialisation en évitant des *opcodes* PUT superflus. Elle ne doit pas être utilisée sur un objet contenant une référence à lui-même, car le `Pickler` entre alors dans une récursion infinie.

Utilisez plutôt `pickletools.optimize()` pour obtenir des données sérialisées plus compactes.

class pickle.**Unpickler**(*file*, *, *fix_imports=True*, *encoding='ASCII'*, *errors='strict'*, *buffers=None*)

Les objets de cette classe sont des désérialiseurs, qui lisent un flux de données pour le convertir en objet.

Il n'y a nul besoin d'argument *protocol*. La version du protocole avec lequel sont encodées les données est déterminée automatiquement.

L'argument *file* doit posséder trois méthodes qui proviennent de l'interface de `io.BufferedIOBase`. Ce sont : `read()`, prenant un entier, `readinto()`, prenant un tampon, et `readline()`, sans arguments. *file* peut donc être aussi bien un fichier sur disque ouvert en mode lecture binaire qu'un objet `io.BytesIO`, ou un objet quelconque vérifiant ces critères.

Les paramètres facultatifs *fix_imports*, *encoding* et *errors* sont dédiés à la compatibilité de la désérialisation avec les flux binaires générés par Python 2. Si *fix_imports* est vrai, *pickle* tente de modifier les anciens noms des modules que l'on trouve en Python 2, pour les remplacer par ceux en usage en Python 3. Les paramètres *encoding* et *errors* contrôlent la façon de décoder les chaînes de caractères 8 bits. Leurs valeurs par défaut respectives sont `'ASCII'` et `'strict'`. *encoding* peut être mis à `'bytes'` pour lire des chaînes d'octets en tant que *bytes*. Il doit être mis à `'latin1'` pour désérialiser des tableaux NumPy ou des instances de

`datetime`, `date` et `time` sérialisées par Python 2.

If *buffers* is `None` (the default), then all data necessary for deserialization must be contained in the pickle stream. This means that the *buffer_callback* argument was `None` when a `Pickler` was instantiated (or when `dump()` or `dumps()` was called).

If *buffers* is not `None`, it should be an iterable of buffer-enabled objects that is consumed each time the pickle stream references an *out-of-band* buffer view. Such buffers have been given in order to the *buffer_callback* of a Pickler object.

Modifié dans la version 3.8 : Ajout de l'argument *buffers*.

load()

Lit la représentation sérialisée d'un objet depuis le fichier ouvert passé au constructeur, et reconstitue l'objet qui y est stocké, avec tous les objets qu'il contient. Les octets situés au-delà de la fin de la représentation binaire sont ignorés.

persistent_load(*pid*)

Par défaut, cette méthode lève une exception `UnpicklingError`.

Si elle est définie autrement dans une sous-classe, `persistent_load()` doit renvoyer l'objet correspondant à l'ID persistant *pid*. Si celui-ci est invalide, elle doit lever une exception `UnpicklingError`.

La section *Persistance d'objets externes* donne des détails et exemples.

Modifié dans la version 3.13 : Add the default implementation of this method in the C implementation of `Unpickler`.

find_class(*module*, *name*)

Importe *module* si besoin, et renvoie l'objet du nom *name* qu'il contient. *module* et *name* sont des chaînes de caractères (classe `str`). Contrairement à ce que son nom laisse penser, `find_class()` est également appelée pour trouver les fonctions.

Les classes filles peuvent redéfinir cette méthode pour restreindre la désérialisation à certains types d'objets ou à d'autres conditions, notamment en vue de réduire les risques de sécurité. Voir *Restriction des noms dans l'espace de nommage global* pour plus de détails.

Lève un *événement d'audit* `pickle.find_class` avec les arguments `module` et `name`.

class pickle.**PickleBuffer**(*buffer*)

Encapsule un objet tampon contenant des données sérialisables. *buffer* doit être un objet prenant en charge le protocole tampon, comme un *objet octet-compatible* ou un tableau n-dimensionnel.

Les objets `PickleBuffer` savent gérer le protocole tampon. Il est donc possible de les passer à d'autres API qui attendent un tampon, comme `memoryview`.

Les objets `PickleBuffer` ne peuvent être sérialisés qu'avec le protocole 5 ou supérieur. Ils sont susceptibles d'être sérialisés *hors-bande*.

Ajouté dans la version 3.8.

raw()

Renvoie une `memoryview` de l'espace mémoire sous-jacent à ce tampon. La *memoryview* renvoyée est unidimensionnelle et C-contiguë. Elle a le format `B` (octets sans signe). `BufferError` est levée si le tampon n'est ni C-contigu, ni Fortran-contigu.

release()

Release the underlying buffer exposed by the PickleBuffer object.

12.1.4 Quels objets sont sérialisables ?

Les objets des types suivants peuvent être sérialisés :

- built-in constants (`None`, `True`, `False`, `Ellipsis`, and `NotImplemented`) ;
- integers, floating-point numbers, complex numbers ;
- strings, bytes, bytearrays ;

— tuples, lists, sets, and dictionaries containing only picklable objects ;
— functions (built-in and user-defined) accessible from the top level of a module (using `def`, not `lambda`) ;
— classes accessible from the top level of a module ;
— instances of such classes whose the result of calling `__getstate__()` is picklable (see section *Sérialisation des instances d'une classe* for details).

Si vous essayez de sérialiser un objet qui ne peut pas l'être, une exception de type `PicklingError` est levée. Lorsque cela se produit, il est possible qu'un certain nombre d'octets aient déjà été écrits dans le fichier ou flux. La sérialisation d'une structure de donnée avec de nombreux niveaux d'imbrication peut lever une exception `RecursionError`. Pour augmenter la limite (avec précaution), voir `sys.setrecursionlimit()`.

Note that functions (built-in and user-defined) are pickled by fully *qualified name*, not by value. [2] This means that only the function name is pickled, along with the name of the containing module and classes. Neither the function's code, nor any of its function attributes are pickled. Thus the defining module must be importable in the unpickling environment, and the module must contain the named object, otherwise an exception will be raised. [3]

Similarly, classes are pickled by fully qualified name, so the same restrictions in the unpickling environment apply. Note that none of the class's code or data is pickled, so in the following example the class attribute `attr` is not restored in the unpickling environment :

```
class Foo:
    attr = 'A class attribute'

picklestring = pickle.dumps(Foo)
```

These restrictions are why picklable functions and classes must be defined at the top level of a module.

Similarly, when class instances are pickled, their class's code and data are not pickled along with them. Only the instance data are pickled. This is done on purpose, so you can fix bugs in a class or add methods to the class and still load objects that were created with an earlier version of the class. If you plan to have long-lived objects that will see many versions of a class, it may be worthwhile to put a version number in the objects so that suitable conversions can be made by the class's `__setstate__()` method.

12.1.5 Sérialisation des instances d'une classe

Dans cette section sont décrits les mécanismes généraux qui s'offrent à vous pour définir, personnaliser et contrôler la manière dont les instances d'une classe sont sérialisées et désérialisées.

In most cases, no additional code is needed to make instances picklable. By default, pickle will retrieve the class and the attributes of an instance via introspection. When a class instance is unpickled, its `__init__()` method is usually *not* invoked. The default behaviour first creates an uninitialized instance and then restores the saved attributes. The following code shows an implementation of this behaviour :

```
def save(obj):
    return (obj.__class__, obj.__dict__)

def restore(cls, attributes):
    obj = cls.__new__(cls)
    obj.__dict__.update(attributes)
    return obj
```

Les classes peuvent personnaliser le comportement par défaut en définissant des méthodes spéciales :

`object.__getnewargs_ex__()`

Dans les protocoles 2 et suivants, les classes peuvent personnaliser les valeurs passées à la méthode `__new__()` lors de la désérialisation. Elles le font en définissant une méthode `__getnewargs_ex__()` qui renvoie un couple `(args, kwargs)`, où *args* est un *n*-uplet des arguments positionnels et *kwargs* un dictionnaire des arguments nommés qui seront passés à `__new__()` — autrement dit, l'appel sera `classe.__new__(*args, **kwargs)`.

2. C'est la raison pour laquelle les fonctions `lambda` ne peuvent pas être sérialisées : elles partagent toutes le même nom, à savoir `<lambda>`.
3. L'exception levée est généralement de type `ImportError` ou `AttributeError`, mais ce n'est pas systématique.

Définissez cette méthode seulement si la méthode `__new__()` de votre classe demande des arguments nommés. Dans le cas contraire, mieux vaut définir `__getnewargs__()` pour préserver la compatibilité avec les protocoles anciens.

Modifié dans la version 3.6 : `__getnewargs_ex__()` est désormais appelée dans les protocoles 2 et 3.

`object.__getnewargs__()`

Comme `__getnewargs_ex__()`, mais ne permet que les arguments positionnels. Cette méthode doit renvoyer le *n*-uplet `args` des arguments passés à `__new__()` lors de la désérialisation : l'appel sera `classe.__new__(*args)`.

Si `__getnewargs_ex__()` est définie, elle prend la priorité et `__getnewargs__()` n'est jamais appelée.

Modifié dans la version 3.6 : Auparavant, `__getnewargs__()` était appelée au lieu de `__getnewargs_ex__()` dans les protocoles 2 et 3.

`object.__getstate__()`

Classes can further influence how their instances are pickled by overriding the method `__getstate__()`. It is called and the returned object is pickled as the contents for the instance, instead of a default state. There are several cases :

— For a class that has no instance `__dict__` and no `__slots__`, the default state is `None`.
— For a class that has an instance `__dict__` and no `__slots__`, the default state is `self.__dict__`.
— For a class that has an instance `__dict__` and `__slots__`, the default state is a tuple consisting of two dictionaries : `self.__dict__`, and a dictionary mapping slot names to slot values. Only slots that have a value are included in the latter.
— For a class that has `__slots__` and no instance `__dict__`, the default state is a tuple whose first item is `None` and whose second item is a dictionary mapping slot names to slot values described in the previous bullet.

Modifié dans la version 3.11 : Added the default implementation of the `__getstate__()` method in the `object` class.

`object.__setstate__(state)`

Lors de la désérialisation, l'état de l'instance est passé à la méthode `__setstate__()`, si elle est définie (l'objet *state* n'a pas besoin d'être un dictionnaire). Si elle ne l'est pas, les attributs de l'objet sont tirés de l'état, qui dans ce cas doit être obligatoirement un dictionnaire.

> **Note**
>
> If `__reduce__()` returns a state with value `None` at pickling, the `__setstate__()` method will not be called upon unpickling.

Refer to the section *Traitement des objets à état* for more information about how to use the methods `__getstate__()` and `__setstate__()`.

> **Note**
>
> At unpickling time, some methods like `__getattr__()`, `__getattribute__()`, or `__setattr__()` may be called upon the instance. In case those methods rely on some internal invariant being true, the type should implement `__new__()` to establish such an invariant, as `__init__()` is not called when unpickling an instance.

As we shall see, pickle does not use directly the methods described above. In fact, these methods are part of the copy protocol which implements the `__reduce__()` special method. The copy protocol provides a unified interface for retrieving the data necessary for pickling and copying objects. [4]

Although powerful, implementing `__reduce__()` directly in your classes is error prone. For this reason, class designers should use the high-level interface (i.e., `__getnewargs_ex__()`, `__getstate__()` and `__setstate__()`) whenever possible. We will show, however, cases where using `__reduce__()` is the only option or leads to more efficient pickling or both.

4. Le module `copy` fait appel à ce protocole pour les opérations de copie superficielle comme récursive.

`object.__reduce__()`

Voici l'interface de la méthode `__reduce__()`. Elle ne prend aucun argument et renvoie soit une chaîne de caractères, soit (c'est conseillé) un *n*-uplet. On appelle souvent l'objet renvoyé « valeur de réduction ».

If a string is returned, the string should be interpreted as the name of a global variable. It should be the object's local name relative to its module ; the pickle module searches the module namespace to determine the object's module. This behaviour is typically useful for singletons.

Si c'est un *n*-uplet qui est renvoyé, ses éléments sont interprétés dans l'ordre comme suit. Les deux premiers éléments sont obligatoires, les quatre suivants sont facultatifs et peuvent être simplement omis, ou bien mis à `None`. Les éléments sont, dans l'ordre :

— Un objet appelable qui sera appelé pour créer l'objet initial.
— Un *n*-uplet d'arguments passés à cet objet appelable. Donnez un *n*-uplet vide si l'objet appelable n'accepte pas d'arguments.
— L'état de l'objet, qui sera passé à la méthode `__setstate__()` comme vu précédemment. Si la méthode n'existe pas, cet élément doit être un dictionnaire, et ses éléments compléteront l'attribut `__dict__`.
— Optionally, an iterator (and not a sequence) yielding successive items. These items will be appended to the object either using `obj.append(item)` or, in batch, using `obj.extend(list_of_items)`. This is primarily used for list subclasses, but may be used by other classes as long as they have *append and extend methods* with the appropriate signature. (Whether `append()` or `extend()` is used depends on which pickle protocol version is used as well as the number of items to append, so both must be supported.)
— Un itérateur (non pas une séquence). Les éléments qu'il fournit doivent être des couples `(clé, valeur)`. Ils sont ajoutés dans l'objet par affectation aux clés : `objet[clé] = valeur`. Ceci est principalement utile aux classes héritant de `dict`, mais peut servir à d'autres classes à la seule condition qu'elles implémentent la méthode `__setitem__()`.
— Un objet appelable qui puisse recevoir en arguments l'objet et son état. Ceci permet de redéfinir le processus de reconstruction des attributs pour un objet en particulier, outrepassant la méthode `__setstate__()`. Si cet objet appelable est fourni, `__setstate__()` n'est pas appelée.

Ajouté dans la version 3.8 : ajout du sixième élément.

`object.__reduce_ex__(protocol)`

Il est également possible de définir une méthode `__reduce_ex__()`. La seule différence est qu'elle prend la version du protocole en argument. Si elle est définie, elle prend le pas sur `__reduce__()`. De plus, `__reduce__()` devient automatiquement un alias pour sa version étendue. Cette méthode est principalement destinée à renvoyer des valeurs de réduction compatibles avec les versions anciennes de Python.

Persistance d'objets externes

Pour les besoins de la persistance, `pickle` permet des références à des objets en dehors du flux sérialisé. Ils sont identifiés par un ID persistant. Le protocole 0 requiert que cet ID soit une chaîne de caractères alphanumériques [5]. Les suivants autorisent un objet quelconque.

`pickle` délègue la résolution des ID à des méthodes définies par l'utilisateur sur les objets sérialiseurs et désérialiseurs, à savoir `persistent_id()` et `persistent_load()`.

Pour affecter à des objets leurs ID persistants provenant d'une source externe, le sérialiseur doit posséder une méthode `persistent_id()` qui prend un objet et renvoie soit `None`, soit son ID. Si cette méthode renvoie `None`, l'objet est sérialisé de la manière habituelle. Si un ID est renvoyé, sous forme de chaîne de caractères, c'est cette chaîne qui est sérialisée et elle est marquée de manière spéciale pour être reconnue comme un ID persistant.

Pour désérialiser des objets identifiés par un ID externe, un désérialiseur doit posséder une méthode `persistent_load()` qui prend un ID et renvoie l'objet qu'il désigne.

Voici un exemple complet qui montre comment sérialiser des objets externes en leur affectant des ID persistants.

```
# Simple example presenting how persistent ID can be used to pickle
# external objects by reference.
```

(suite sur la page suivante)

5. The limitation on alphanumeric characters is due to the fact that persistent IDs in protocol 0 are delimited by the newline character. Therefore if any kind of newline characters occurs in persistent IDs, the resulting pickled data will become unreadable.

(suite de la page précédente)

```
import pickle
import sqlite3
from collections import namedtuple

# Simple class representing a record in our database.
MemoRecord = namedtuple("MemoRecord", "key, task")

class DBPickler(pickle.Pickler):

    def persistent_id(self, obj):
        # Instead of pickling MemoRecord as a regular class instance, we emit a
        # persistent ID.
        if isinstance(obj, MemoRecord):
            # Here, our persistent ID is simply a tuple, containing a tag and a
            # key, which refers to a specific record in the database.
            return ("MemoRecord", obj.key)
        else:
            # If obj does not have a persistent ID, return None. This means obj
            # needs to be pickled as usual.
            return None


class DBUnpickler(pickle.Unpickler):

    def __init__(self, file, connection):
        super().__init__(file)
        self.connection = connection

    def persistent_load(self, pid):
        # This method is invoked whenever a persistent ID is encountered.
        # Here, pid is the tuple returned by DBPickler.
        cursor = self.connection.cursor()
        type_tag, key_id = pid
        if type_tag == "MemoRecord":
            # Fetch the referenced record from the database and return it.
            cursor.execute("SELECT * FROM memos WHERE key=?", (str(key_id),))
            key, task = cursor.fetchone()
            return MemoRecord(key, task)
        else:
            # Always raises an error if you cannot return the correct object.
            # Otherwise, the unpickler will think None is the object referenced
            # by the persistent ID.
            raise pickle.UnpicklingError("unsupported persistent object")


def main():
    import io
    import pprint

    # Initialize and populate our database.
    conn = sqlite3.connect(":memory:")
    cursor = conn.cursor()
    cursor.execute("CREATE TABLE memos(key INTEGER PRIMARY KEY, task TEXT)")
    tasks = (
        'give food to fish',
        'prepare group meeting',
```

(suite sur la page suivante)

(suite de la page précédente)

```
        'fight with a zebra',
        )
    for task in tasks:
        cursor.execute("INSERT INTO memos VALUES(NULL, ?)", (task,))

    # Fetch the records to be pickled.
    cursor.execute("SELECT * FROM memos")
    memos = [MemoRecord(key, task) for key, task in cursor]
    # Save the records using our custom DBPickler.
    file = io.BytesIO()
    DBPickler(file).dump(memos)

    print("Pickled records:")
    pprint.pprint(memos)

    # Update a record, just for good measure.
    cursor.execute("UPDATE memos SET task='learn italian' WHERE key=1")

    # Load the records from the pickle data stream.
    file.seek(0)
    memos = DBUnpickler(file, conn).load()

    print("Unpickled records:")
    pprint.pprint(memos)


if __name__ == '__main__':
    main()
```

Tables de distribution

Pour personnaliser la sérialisation d'une classe à un endroit particulier sans affecter le reste du code, on peut créer un sérialiseur avec une table de distribution spécifique.

The global dispatch table managed by the *copyreg* module is available as `copyreg.dispatch_table`. Therefore, one may choose to use a modified copy of `copyreg.dispatch_table` as a private dispatch table.

Par exemple, le code

```
f = io.BytesIO()
p = pickle.Pickler(f)
p.dispatch_table = copyreg.dispatch_table.copy()
p.dispatch_table[SomeClass] = reduce_SomeClass
```

crée une instance de la classe *pickle.Pickler* avec une table de distribution propre qui traite la classe `SomeClass` de manière spécifique. Le code

```
class MyPickler(pickle.Pickler):
    dispatch_table = copyreg.dispatch_table.copy()
    dispatch_table[SomeClass] = reduce_SomeClass
f = io.BytesIO()
p = MyPickler(f)
```

does the same but all instances of `MyPickler` will by default share the private dispatch table. On the other hand, the code

```
copyreg.pickle(SomeClass, reduce_SomeClass)
f = io.BytesIO()
p = pickle.Pickler(f)
```

modifies the global dispatch table shared by all users of the *copyreg* module.

Traitement des objets à état

Here's an example that shows how to modify pickling behavior for a class. The `TextReader` class below opens a text file, and returns the line number and line contents each time its `readline()` method is called. If a `TextReader` instance is pickled, all attributes *except* the file object member are saved. When the instance is unpickled, the file is reopened, and reading resumes from the last location. The `__setstate__()` and `__getstate__()` methods are used to implement this behavior.

```
class TextReader:
    """Print and number lines in a text file."""

    def __init__(self, filename):
        self.filename = filename
        self.file = open(filename)
        self.lineno = 0

    def readline(self):
        self.lineno += 1
        line = self.file.readline()
        if not line:
            return None
        if line.endswith('\n'):
            line = line[:-1]
        return "%i: %s" % (self.lineno, line)

    def __getstate__(self):
        # Copy the object's state from self.__dict__ which contains
        # all our instance attributes. Always use the dict.copy()
        # method to avoid modifying the original state.
        state = self.__dict__.copy()
        # Remove the unpicklable entries.
        del state['file']
        return state

    def __setstate__(self, state):
        # Restore instance attributes (i.e., filename and lineno).
        self.__dict__.update(state)
        # Restore the previously opened file's state. To do so, we need to
        # reopen it and read from it until the line count is restored.
        file = open(self.filename)
        for _ in range(self.lineno):
            file.readline()
        # Finally, save the file.
        self.file = file
```

Voici un exemple d'utilisation :

```
>>> reader = TextReader("hello.txt")
>>> reader.readline()
'1: Hello world!'
>>> reader.readline()
```

(suite sur la page suivante)

(suite de la page précédente)

```
'2: I am line number two.'
>>> new_reader = pickle.loads(pickle.dumps(reader))
>>> new_reader.readline()
'3: Goodbye!'
```

12.1.6 Réduction personnalisée pour les types, fonctions et autres objets

Ajouté dans la version 3.8.

Parfois, la simple utilisation de `dispatch_table` n'offre pas assez de flexibilité. On peut vouloir changer la méthode de sérialisation selon d'autres critères que le type de l'objet, ou bien personnaliser la sérialisation des fonctions et des classes.

For those cases, it is possible to subclass from the `Pickler` class and implement a `reducer_override()` method. This method can return an arbitrary reduction tuple (see `__reduce__()`). It can alternatively return `NotImplemented` to fallback to the traditional behavior.

Si `dispatch_table` et `reducer_override()` sont tous les deux définis, `reducer_override()` a la priorité.

Note

Pour des raisons de performance, la méthode `reducer_override()` n'est jamais appelée sur `None`, `True`, `False`, ainsi que les instances exactes (pas dérivées) de `int`, `float`, `bytes`, `str`, `dict`, `set`, `frozenset`, `list` et `tuple`.

Voici un exemple simple qui implémente la sérialisation d'une classe :

```
import io
import pickle

class MyClass:
    my_attribute = 1

class MyPickler(pickle.Pickler):
    def reducer_override(self, obj):
        """Custom reducer for MyClass."""
        if getattr(obj, "__name__", None) == "MyClass":
            return type, (obj.__name__, obj.__bases__,
                          {'my_attribute': obj.my_attribute})
        else:
            # For any other object, fallback to usual reduction
            return NotImplemented

f = io.BytesIO()
p = MyPickler(f)
p.dump(MyClass)

del MyClass

unpickled_class = pickle.loads(f.getvalue())

assert isinstance(unpickled_class, type)
assert unpickled_class.__name__ == "MyClass"
assert unpickled_class.my_attribute == 1
```

12.1.7 Tampons hors-bande

Ajouté dans la version 3.8.

Le module `pickle` est parfois utilisé pour transférer des quantités énormes de données. Il peut devenir important de réduire les copies de mémoire au minimum pour préserver la performance et diminuer l'usage des ressources matérielles. Cependant, dans son contexte courant d'utilisation, le module `pickle` effectue des copies depuis et vers le flux de données pour les besoins de la conversion de structures d'objets semblables à des graphes en flux séquentiels d'octets.

Cette contrainte peut être levée si le *producteur* (qui implémente les types d'objets à transférer) et le *consommateur* (qui implémente le système de communication) emploient les possibilités de transfert hors-bande offertes par les protocoles 5 et suivants.

API des producteurs

The large data objects to be pickled must implement a `__reduce_ex__()` method specialized for protocol 5 and higher, which returns a `PickleBuffer` instance (instead of e.g. a `bytes` object) for any large data.

Les objets `PickleBuffer` ne font que signaler que leur tampon permet le transfert hors-bande. Ils demeurent compatibles avec l'utilisation classique du module `pickle`. Cependant, les consommateurs peuvent aussi choisir d'indiquer à `pickle` qu'ils gèrent eux-mêmes ces tampons.

API des consommateurs

Un système de communication peut gérer de manière spécifique les objets `PickleBuffer` générés lors de la sérialisation d'un réseau d'objets.

Du côté de l'expéditeur, il faut passer le paramètre *buffer_callback* à `Pickler` (ou à `dump()` ou `dumps()`). Le *buffer_callback* sera appelé avec chaque `PickleBuffer` généré lors de la sérialisation du réseau d'objets. Les tampons accumulés par le *buffer_callback* ne verront pas leurs données copiées dans le flux sérialisé. Il leur sera substitué un marqueur léger.

Du côté du receveur, il faut passer l'argument *buffers* à `Unpickler` (ou `load()` ou bien `loads()`). *buffers* est un itérable des tampons passés à *buffer_callback*. Il doit fournir les tampons dans le même ordre que celui dans lequel ils ont été passés à *buffer_callback*. Les tampons fournis constituent la source des données qu'attendent les reconstructeurs des objets dont la sérialisation a abouti aux objets `PickleBuffer`.

Entre expéditeur et receveur, le système de communication peut implémenter son propre mécanisme de transfert pour les tampons hors-bande. Parmi les optimisations possibles se trouvent l'utilisation de mémoire partagée et la compression spécifique au type de données.

Exemple

Voici un exemple trivial où est implémentée une classe fille de `bytearray` capable de sérialisation hors-bande :

```
class ZeroCopyByteArray(bytearray):

    def __reduce_ex__(self, protocol):
        if protocol >= 5:
            return type(self)._reconstruct, (PickleBuffer(self),), None
        else:
            # PickleBuffer is forbidden with pickle protocols <= 4.
            return type(self)._reconstruct, (bytearray(self),)

    @classmethod
    def _reconstruct(cls, obj):
        with memoryview(obj) as m:
            # Get a handle over the original buffer object
            obj = m.obj
            if type(obj) is cls:
                # Original buffer object is a ZeroCopyByteArray, return it
```

(suite sur la page suivante)

(suite de la page précédente)

```
                # as-is.
                return obj
            else:
                return cls(obj)
```

Lorsqu'il rencontre le bon type, le reconstructeur (la méthode de classe `_reconstruct`) renvoie directement le tampon original. Il s'agit d'une manière simple de simuler l'absence de copie dans cet exemple simpliste.

En tant que consommateur des objets, on peut les sérialiser de la manière classique. La désérialisation conduit alors à une copie :

```
b = ZeroCopyByteArray(b"abc")
data = pickle.dumps(b, protocol=5)
new_b = pickle.loads(data)
print(b == new_b)  # True
print(b is new_b)  # False: a copy was made
```

Mais en passant un *buffer_callback* et en donnant les tampons accumulés au désérialiseur, il n'y a plus de copie :

```
b = ZeroCopyByteArray(b"abc")
buffers = []
data = pickle.dumps(b, protocol=5, buffer_callback=buffers.append)
new_b = pickle.loads(data, buffers=buffers)
print(b == new_b)  # True
print(b is new_b)  # True: no copy was made
```

Cet exemple est limité par le fait que `bytearray` effectue sa propre allocation de mémoire. Il n'est pas possible de créer un `bytearray` sur la mémoire d'un autre objet. Cependant, certains types de données que l'on trouve dans des bibliothèques externes, comme les tableaux NumPy, n'ont pas cette limitation. Le passage hors-bande permet alors de n'effectuer aucune copie (ou bien de minimiser le nombre de copies) lors du transfert de données d'un système à l'autre ou d'un processus à l'autre.

Voir aussi

PEP 574 — Protocole *pickle* 5 avec données hors-bande

12.1.8 Restriction des noms dans l'espace de nommage global

Par défaut, la désérialisation importe toutes les classes ou fonctions que demande le flux de données. Dans bien des cas, ce comportement est inacceptable, puisqu'il permet de faire exécuter du code arbitraire dans l'environnement de désérialisation. Observez le résultat de ce flux de données fait-main lorsqu'il est lu :

```
>>> import pickle
>>> pickle.loads(b"cos\nsystem\n(S'echo hello world'\ntR.")
hello world
0
```

Dans cet exemple, le désérialiseur importe la fonction `os.system()` et l'applique à la chaîne de caractères `"echo hello world"`. C'est inoffensif, mais il n'est pas difficile d'imaginer des variantes qui endommageraient le système.

C'est pour cette raison qu'il s'avère parfois nécessaire de contrôler ce qui peut être désérialisé. Cela est possible en redéfinissant la méthode `Unpickler.find_class()`. Contrairement à ce que son nom laisse penser, `Unpickler.find_class()` est appelée pour tous les noms à chercher dans l'espace de nommage global, ce qui inclut les classes mais aussi les fonctions. Par ce biais, il est possible d'interdire complètement la résolution des noms globaux ou de la restreindre à un sous-ensemble que l'on considère sûr.

Voici un exemple de désérialiseur qui permet seulement la désérialisation d'un petit nombre de classes sûres du module `builtins` :

```
import builtins
import io
import pickle

safe_builtins = {
    'range',
    'complex',
    'set',
    'frozenset',
    'slice',
}

class RestrictedUnpickler(pickle.Unpickler):

    def find_class(self, module, name):
        # Only allow safe classes from builtins.
        if module == "builtins" and name in safe_builtins:
            return getattr(builtins, name)
        # Forbid everything else.
        raise pickle.UnpicklingError("global '%s.%s' is forbidden" %
                                     (module, name))

def restricted_loads(s):
    """Helper function analogous to pickle.loads()."""
    return RestrictedUnpickler(io.BytesIO(s)).load()
```

A sample usage of our unpickler working as intended :

```
>>> restricted_loads(pickle.dumps([1, 2, range(15)]))
[1, 2, range(0, 15)]
>>> restricted_loads(b"cos\nsystem\n(S'echo hello world'\ntR.")
Traceback (most recent call last):
  ...
pickle.UnpicklingError: global 'os.system' is forbidden
>>> restricted_loads(b'cbuiltins\neval\n'
...                  b'(S\'getattr(__import__("os"), "system")'
...                  b'("echo hello world")\'\ntR.')
Traceback (most recent call last):
  ...
pickle.UnpicklingError: global 'builtins.eval' is forbidden
```

Comme le montre l'exemple, il faut faire attention aux objets que l'on autorise à être désérialisés. Si la sécurité est une priorité, il peut être sage de se tourner vers des alternatives comme l'API du module `xmlrpc.client`, ou des bibliothèques tierces.

12.1.9 Performances

Recent versions of the pickle protocol (from protocol 2 and upwards) feature efficient binary encodings for several common features and built-in types. Also, the `pickle` module has a transparent optimizer written in C.

12.1.10 Exemples

Dans les cas les plus simples, utilisez les fonctions `dump()` et `load()`.

```
import pickle

# An arbitrary collection of objects supported by pickle.
```

(suite sur la page suivante)

(suite de la page précédente)

```
data = {
    'a': [1, 2.0, 3+4j],
    'b': ("character string", b"byte string"),
    'c': {None, True, False}
}

with open('data.pickle', 'wb') as f:
    # Pickle the 'data' dictionary using the highest protocol available.
    pickle.dump(data, f, pickle.HIGHEST_PROTOCOL)
```

Le code suivant lit les données qui viennent d'être sérialisées

```
import pickle

with open('data.pickle', 'rb') as f:
    # The protocol version used is detected automatically, so we do not
    # have to specify it.
    data = pickle.load(f)
```

Voir aussi

Module ***copyreg***
Enregistre les fonctions de sérialisation pour les types définis par l'utilisateur.

Module ***pickletools***
Outils pour travailler sur les données sérialisées et les analyser.

Module ***shelve***
Bases de données indexées (module fondé sur `pickle`).

Module ***copy***
Copie superficielle ou récursive d'objets.

Module ***marshal***
Sérialisation haute-performance des types natifs.

Notes

12.2 `copyreg` --- Register `pickle` support functions

Code source : Lib/copyreg.py

Le module `copyreg` permet de définir des fonctions utilisées durant la sérialisation avec *pickle* de certains objets. Les modules `pickle` et `copy` utilisent ces fonctions lors d'une sérialisation ou d'une copie de ces objets. Le module propose alors des informations de configuration à propos de constructeurs d'objets qui ne sont pas des classes. De tels constructeurs peuvent être des instances de classes ou des fonctions.

`copyreg.`**`constructor`**(*object*)

Déclare *object* comme étant un constructeur valide. Si *object* n'est pas appelable (et n'est donc pas un constructeur valide), l'erreur `TypeError` est levée.

`copyreg.`**`pickle`**(*type*, *function*, *constructor_ob=None*)

Declares that *function* should be used as a "reduction" function for objects of type *type*. *function* must return either a string or a tuple containing between two and six elements. See the `dispatch_table` for more details on the interface of *function*.

The *constructor_ob* parameter is a legacy feature and is now ignored, but if passed it must be a callable.

Note that the `dispatch_table` attribute of a pickler object or subclass of `pickle.Pickler` can also be used for declaring reduction functions.

12.2.1 Exemple

L'exemple si-dessous essaye de démontrer comment enregistrer une fonction *pickle* et comment elle sera utilisée :

```
>>> import copyreg, copy, pickle
>>> class C:
...     def __init__(self, a):
...         self.a = a
...
>>> def pickle_c(c):
...     print("pickling a C instance...")
...     return C, (c.a,)
...
>>> copyreg.pickle(C, pickle_c)
>>> c = C(1)
>>> d = copy.copy(c)
pickling a C instance...
>>> p = pickle.dumps(c)
pickling a C instance...
```

12.3 `shelve` --- Python object persistence

Code source : Lib/shelve.py

Une étagère (*shelf* en anglais) est un objet persistant similaire à un dictionnaire. La différence avec les bases de données *dbm* est que les valeurs (pas les clés !) dans une étagère peuvent être des objets Python arbitraires — n'importe quoi que le module `pickle` peut gérer. Cela inclut la plupart des instances de classe, des types de données récursives, et les objets contenant beaucoup de sous-objets partagés. Les clés sont des chaînes de caractères ordinaires.

`shelve.open`(*filename*, *flag='c'*, *protocol=None*, *writeback=False*)

Ouvre un dictionnaire persistant. Le nom de fichier spécifié est le nom de fichier sans (son) extension pour la base de données sous-jacente. Comme effet de bord, une extension peut être ajoutée au nom de fichier et plus d'un fichier peut être créé. Par défaut, le fichier de base de données sous-jacente est ouvert en lecture et en écriture. Le paramètre optionnel *flag* possède la même interprétation que le paramètre *flag* de `dbm.open()`.

By default, pickles created with `pickle.DEFAULT_PROTOCOL` are used to serialize values. The version of the pickle protocol can be specified with the *protocol* parameter.

À cause de la sémantique de Python, une étagère ne peut pas savoir lorsqu'une entrée mutable de dictionnaire persistant est modifiée. Par défaut les objets modifiés sont écrits *seulement* lorsqu'ils sont assignés à une étagère (voir *Exemple*). Si le paramètre optionnel *writeback* est mis à `True`, toutes les entrées déjà accédées sont aussi mises en cache en mémoire, et ré-écrites sur `sync()` et `close()` ; cela peut faciliter la modification des entrées modifiables dans le dictionnaire persistant, mais, si vous accédez à beaucoup d'entrées, cela peut consommer beaucoup de mémoire cache, et cela peut rendre l'opération de fermeture très lente puisque toutes les entrées déjà accédées sont ré-écrites (il n'y a aucun moyen de savoir quelles entrées déjà accédées sont mutables, ni lesquelles ont été vraiment modifiées).

Modifié dans la version 3.10 : `pickle.DEFAULT_PROTOCOL` is now used as the default pickle protocol.

Modifié dans la version 3.11 : Accepts *path-like object* for filename.

> **Note**
>
> Ne pas se fier à la fermeture automatique de l'étagère ; appelez toujours `close()` explicitement quand vous n'en avez plus besoin, ou utilisez `shelve.open()` comme un gestionnaire de contexte :

```
with shelve.open('spam') as db:
    db['eggs'] = 'eggs'
```

Avertissement

Puisque le module `shelve` utilise en arrière plan `pickle`, il n'est pas sûr de charger une étagère depuis une source non fiable. Comme avec *pickle*, charger une étagère peut exécuter du code arbitraire.

Les objets d'étagère gèrent la plupart des méthodes et opérations des dictionnaires (à l'exception de la copie, des constructeurs et des opérateurs `|` et `|=`). Cela facilite la transition depuis les scripts utilisant des dictionnaires vers ceux nécessitant un stockage persistant.

Deux méthodes supplémentaires sont autorisées :

`Shelf.sync()`

Réécrit toutes les entrées dans le cache si l'étagère a été ouverte avec *writeback* passé à `True`. Vide le cache et synchronise le dictionnaire persistant sur le disque, si faisable. Elle est appelée automatiquement quand l'étagère est fermée avec `close()`.

`Shelf.close()`

Synchronise et ferme l'objet *dict* persistant. Les opérations sur une étagère fermée échouent avec une `ValueError`.

Voir aussi

Persistent dictionary recipe with widely supported storage formats and having the speed of native dictionaries.

12.3.1 Limites

— Le choix du paquet de base de données à utiliser (comme `dbm.ndbm` ou `dbm.gnu`) dépend de l'interface disponible. Donc c'est risqué d'ouvrir la base de données directement en utilisant `dbm`. La base de données est également (malheureusement) sujette à des limitations de `dbm`, si c'est utilisé — cela signifie que (la représentation sérialisée de) l'objet stocké dans la base de données doit être assez petit et, dans de rares cas, des collisions de clés peuvent entraîner le refus de mises à jour de la base de données.

— Le module `shelve` ne gère pas l'accès *concurrent* en lecture/écriture sur les objets stockés (les accès simultanés en lecture sont sûrs). Lorsqu'un programme a une étagère ouverte en écriture, aucun autre programme ne doit l'avoir ouverte en écriture ou lecture. Le verrouillage des fichiers Unix peut être utilisé pour résoudre ce problème, mais cela dépend de la version Unix et nécessite des connaissances à propos de l'implémentation de la base de données utilisée.

— On macOS `dbm.ndbm` can silently corrupt the database file on updates, which can cause hard crashes when trying to read from the database.

class `shelve.Shelf`(*dict*, *protocol=None*, *writeback=False*, *keyencoding='utf-8'*)

Sous-classe de `collections.abc.MutableMapping` qui stocke les valeurs sérialisées dans l'objet *dict*.

By default, pickles created with `pickle.DEFAULT_PROTOCOL` are used to serialize values. The version of the pickle protocol can be specified with the *protocol* parameter. See the `pickle` documentation for a discussion of the pickle protocols.

Si le paramètre *writeback* est `True`, l'objet garde en cache toutes les entrées accédées et les écrit dans le *dict* aux moments de synchronisation et de fermeture. Cela permet des opérations naturelles sur les entrées mutables, mais peut consommer beaucoup plus de mémoire et rendre les temps de synchronisation et de fermeture très longs.

Le paramètre *keyencoding* est l'encodage utilisé pour encoder les clés avant qu'elles soient utilisées avec le dictionnaire sous-jacent.

Un objet `Shelf` peut également être utilisé comme un gestionnaire de contexte ; il est automatiquement fermé lorsque le bloc `with` se termine.

Modifié dans la version 3.2 : Ajout du paramètre *keyencoding* ; précédemment, les clés étaient toujours encodées en UTF-8.

Modifié dans la version 3.4 : Ajout de la gestion des gestionnaires de contexte.

Modifié dans la version 3.10 : `pickle.DEFAULT_PROTOCOL` is now used as the default pickle protocol.

class shelve.**BsdDbShelf**(*dict*, *protocol=None*, *writeback=False*, *keyencoding='utf-8'*)

A subclass of `Shelf` which exposes `first()`, `next()`, `previous()`, `last()` and `set_location()` methods. These are available in the third-party `bsddb` module from pybsddb but not in other database modules. The *dict* object passed to the constructor must support those methods. This is generally accomplished by calling one of `bsddb.hashopen()`, `bsddb.btopen()` or `bsddb.rnopen()`. The optional *protocol*, *writeback*, and *keyencoding* parameters have the same interpretation as for the `Shelf` class.

class shelve.**DbfilenameShelf**(*filename*, *flag='c'*, *protocol=None*, *writeback=False*)

Sous-classe de `Shelf` qui accepte un *filename* au lieu d'un objet dictionnaire-compatible. Le fichier sous-jacent est ouvert avec `dbm.open()`. Par défaut le fichier est créé en lecture et en écriture. Le paramètre optionnel *flag* peut être interprété de la même manière que pour la fonction `open()`. Les paramètres optionnels *protocol* et *writeback* s'interprètent de la même manière que pour la classe `Shelf`.

12.3.2 Exemple

Pour résumer l'interface (`key` est une chaîne de caractère, `data` est un objet arbitraire) :

```
import shelve

d = shelve.open(filename)  # open -- file may get suffix added by low-level
                           # library

d[key] = data              # store data at key (overwrites old data if
                           # using an existing key)
data = d[key]              # retrieve a COPY of data at key (raise KeyError
                           # if no such key)
del d[key]                 # delete data stored at key (raises KeyError
                           # if no such key)

flag = key in d            # true if the key exists
klist = list(d.keys())     # a list of all existing keys (slow!)

# as d was opened WITHOUT writeback=True, beware:
d['xx'] = [0, 1, 2]        # this works as expected, but...
d['xx'].append(3)          # *this doesn't!* -- d['xx'] is STILL [0, 1, 2]!

# having opened d without writeback=True, you need to code carefully:
temp = d['xx']             # extracts the copy
temp.append(5)             # mutates the copy
d['xx'] = temp             # stores the copy right back, to persist it

# or, d=shelve.open(filename,writeback=True) would let you just code
# d['xx'].append(5) and have it work as expected, BUT it would also
# consume more memory and make the d.close() operation slower.

d.close()                  # close it
```

Voir aussi

Module *dbm*
Interface générique de base de données dans le style de dbm.

Module *pickle*
Sérialisation d'objet utilisé par *shelve*.

12.4 marshal --- Internal Python object serialization

This module contains functions that can read and write Python values in a binary format. The format is specific to Python, but independent of machine architecture issues (e.g., you can write a Python value to a file on a PC, transport the file to a Mac, and read it back there). Details of the format are undocumented on purpose ; it may change between Python versions (although it rarely does). [1]

This is not a general "persistence" module. For general persistence and transfer of Python objects through RPC calls, see the modules *pickle* and *shelve*. The *marshal* module exists mainly to support reading and writing the "pseudo-compiled" code for Python modules of .pyc files. Therefore, the Python maintainers reserve the right to modify the marshal format in backward incompatible ways should the need arise. The format of code objects is not compatible between Python versions, even if the version of the format is the same. De-serializing a code object in the incorrect Python version has undefined behavior. If you're serializing and de-serializing Python objects, use the *pickle* module instead -- the performance is comparable, version independence is guaranteed, and pickle supports a substantially wider range of objects than marshal.

Avertissement

N'utilisez pas le module *marshal* pour lire des données erronées ou malveillantes. Ne démantelez jamais des données reçues d'une source non fiable ou non authentifiée.

Il existe des fonctions de lecture-écriture de fichiers ainsi que des fonctions opérant sur des objets octet.

Not all Python object types are supported ; in general, only objects whose value is independent from a particular invocation of Python can be written and read by this module. The following types are supported :

- Numeric types : *int*, *bool*, *float*, *complex*.
- Strings (*str*) and *bytes*. *Bytes-like objects* like *bytearray* are marshalled as bytes.
- Containers : *tuple*, *list*, *set*, *frozenset*, and (since *version* 5), *slice*. It should be understood that these are supported only if the values contained therein are themselves supported. Recursive containers are supported since *version* 3.
- The singletons *None*, *Ellipsis* and *StopIteration*.
- *code* objects, if *allow_code* is true. See note above about version dependence.

Modifié dans la version 3.4 :

- Added format version 3, which supports marshalling recursive lists, sets and dictionaries.
- Added format version 4, which supports efficient representations of short strings.

Modifié dans la version 3.14 : Added format version 5, which allows marshalling slices.

Le module définit ces fonctions :

marshal.**dump**(*value*, *file*, *version=version*, /, *, *allow_code=True*)

Écrit la valeur sur le fichier ouvert. La valeur doit être un type pris en charge. Le fichier doit être un *fichier binaire* ouvert en écriture.

If the value has (or contains an object that has) an unsupported type, a *ValueError* exception is raised --- but garbage data will also be written to the file. The object will not be properly read back by *load()*. Code objects are only supported if *allow_code* is true.

1. Le nom de ce module provient d'un peu de terminologie utilisée par les concepteurs de Modula-3 (entre autres), qui utilisent le terme *marshalling* pour l'envoi de données sous une forme autonome. À proprement parler, *to marshal* signifie convertir certaines données d'une forme interne à une forme externe (dans une mémoire tampon RPC par exemple) et *unmarshalling* désigne le processus inverse.

L'argument *version* indique le format de données que le `dump` doit utiliser (voir ci-dessous).

Raises an *auditing event* `marshal.dumps` with arguments `value`, `version`.

Modifié dans la version 3.13 : Added the *allow_code* parameter.

`marshal.load`(*file*, /, *, *allow_code=True*)

Read one value from the open file and return it. If no valid value is read (e.g. because the data has a different Python version's incompatible marshal format), raise `EOFError`, `ValueError` or `TypeError`. Code objects are only supported if *allow_code* is true. The file must be a readable *binary file*.

Raises an *auditing event* `marshal.load` with no arguments.

> **Note**
>
> Si un objet contenant un type non pris en charge a été dé-compilé avec `dump()`, `load()` remplacera le type non « dé-compilable » par `None`.

Modifié dans la version 3.10 : This call used to raise a `code.__new__` audit event for each code object. Now it raises a single `marshal.load` event for the entire load operation.

Modifié dans la version 3.13 : Added the *allow_code* parameter.

`marshal.dumps`(*value*, *version=version*, /, *, *allow_code=True*)

Return the bytes object that would be written to a file by `dump(value, file)`. The value must be a supported type. Raise a `ValueError` exception if value has (or contains an object that has) an unsupported type. Code objects are only supported if *allow_code* is true.

L'argument *version* indique le format de données que `dumps` doivent utiliser (voir ci-dessous).

Raises an *auditing event* `marshal.dumps` with arguments `value`, `version`.

Modifié dans la version 3.13 : Added the *allow_code* parameter.

`marshal.loads`(*bytes*, /, *, *allow_code=True*)

Convert the *bytes-like object* to a value. If no valid value is found, raise `EOFError`, `ValueError` or `TypeError`. Code objects are only supported if *allow_code* is true. Extra bytes in the input are ignored.

Raises an *auditing event* `marshal.loads` with argument `bytes`.

Modifié dans la version 3.10 : This call used to raise a `code.__new__` audit event for each code object. Now it raises a single `marshal.loads` event for the entire load operation.

Modifié dans la version 3.13 : Added the *allow_code* parameter.

De plus, les constantes suivantes sont définies :

`marshal.version`

Indicates the format that the module uses. Version 0 is the historical first version ; subsequent versions add new features. Generally, a new version becomes the default when it is introduced.

Version	Available since	New features
1	Python 2.4	Sharing interned strings
2	Python 2.5	Binary representation of floats
3	Python 3.4	Support for object instancing and recursion
4	Python 3.4	Efficient representation of short strings
5	Python 3.14	Support for `slice` objects

Notes

12.5 dbm --- Interfaces to Unix "databases"

Source code : Lib/dbm/__init__.py

---

dbm is a generic interface to variants of the DBM database :

- *dbm.sqlite3*
- *dbm.gnu*
- *dbm.ndbm*

If none of these modules are installed, the slow-but-simple implementation in module *dbm.dumb* will be used. There is a third party interface to the Oracle Berkeley DB.

exception dbm.**error**

A tuple containing the exceptions that can be raised by each of the supported modules, with a unique exception also named *dbm.error* as the first item --- the latter is used when *dbm.error* is raised.

dbm.**whichdb**(*filename*)

This function attempts to guess which of the several simple database modules available --- *dbm.sqlite3*, *dbm.gnu*, *dbm.ndbm*, or *dbm.dumb* --- should be used to open a given file.

Return one of the following values :

- `None` if the file can't be opened because it's unreadable or doesn't exist
- the empty string (`''`) if the file's format can't be guessed
- a string containing the required module name, such as `'dbm.ndbm'` or `'dbm.gnu'`

Modifié dans la version 3.11 : *filename* accepts a *path-like object*.

dbm.**open**(*file*, *flag='r'*, *mode=0o666*)

Open a database and return the corresponding database object.

Paramètres

- **file** (*path-like object*) -- The database file to open.

  If the database file already exists, the *whichdb()* function is used to determine its type and the appropriate module is used ; if it does not exist, the first submodule listed above that can be imported is used.
- **flag** (`str`) --
  - `'r'` (default) : Open existing database for reading only.
  - `'w'` : Open existing database for reading and writing.
  - `'c'` : Open database for reading and writing, creating it if it doesn't exist.
  - `'n'` : Always create a new, empty database, open for reading and writing.
- **mode** (`int`) -- The Unix file access mode of the file (default : octal `0o666`), used only when the database has to be created.

Modifié dans la version 3.11 : *file* accepts a *path-like object*.

The object returned by *open()* supports the same basic functionality as a *dict* ; keys and their corresponding values can be stored, retrieved, and deleted, and the `in` operator and the `keys()` method are available, as well as `get()` and `setdefault()` methods.

Key and values are always stored as *bytes*. This means that when strings are used they are implicitly converted to the default encoding before being stored.

These objects also support being used in a `with` statement, which will automatically close them when done.

Modifié dans la version 3.2 : `get()` and `setdefault()` methods are now available for all *dbm* backends.

Modifié dans la version 3.4 : Added native support for the context management protocol to the objects returned by *open()*.

Modifié dans la version 3.8 : Deleting a key from a read-only database raises a database module specific exception instead of *KeyError*.

The following example records some hostnames and a corresponding title, and then prints out the contents of the database :

```
import dbm

# Open database, creating it if necessary.
with dbm.open('cache', 'c') as db:

    # Record some values
    db[b'hello'] = b'there'
    db['www.python.org'] = 'Python Website'
    db['www.cnn.com'] = 'Cable News Network'

    # Note that the keys are considered bytes now.
    assert db[b'www.python.org'] == b'Python Website'
    # Notice how the value is now in bytes.
    assert db['www.cnn.com'] == b'Cable News Network'

    # Often-used methods of the dict interface work too.
    print(db.get('python.org', b'not present'))

    # Storing a non-string key or value will raise an exception (most
    # likely a TypeError).
    db['www.yahoo.com'] = 4

# db is automatically closed when leaving the with statement.
```

Voir aussi

Module ***shelve***
Persistence module which stores non-string data.

The individual submodules are described in the following sections.

12.5.1 `dbm.sqlite3` --- SQLite backend for dbm

Ajouté dans la version 3.13.

Source code : Lib/dbm/sqlite3.py

This module uses the standard library *sqlite3* module to provide an SQLite backend for the *dbm* module. The files created by *dbm.sqlite3* can thus be opened by *sqlite3*, or any other SQLite browser, including the SQLite CLI.

Availability : not WASI.

This module does not work or is not available on WebAssembly. See *Plateformes WebAssembly* for more information.

`dbm.sqlite3.open`(*filename*, */*, *flag='r'*, *mode=0o666*)

Open an SQLite database. The returned object behaves like a *mapping*, implements a `close()` method, and supports a "closing" context manager via the `with` keyword.

Paramètres

- **filename** (*path-like object*) -- The path to the database to be opened.
- **flag** (`str`) --
  - `'r'` (default) : Open existing database for reading only.
  - `'w'` : Open existing database for reading and writing.
  - `'c'` : Open database for reading and writing, creating it if it doesn't exist.
  - `'n'` : Always create a new, empty database, open for reading and writing.

— **mode** -- The Unix file access mode of the file (default : octal `0o666`), used only when the database has to be created.

12.5.2 `dbm.gnu` --- GNU database manager

Source code : Lib/dbm/gnu.py

The *dbm.gnu* module provides an interface to the GDBM (GNU dbm) library, similar to the *dbm.ndbm* module, but with additional functionality like crash tolerance.

> **Note**
>
> The file formats created by *dbm.gnu* and *dbm.ndbm* are incompatible and can not be used interchangeably.

Availability : not Android, not iOS, not WASI.

This module is not supported on *mobile platforms* or *WebAssembly platforms*.

exception `dbm.gnu.`**error**

Raised on *dbm.gnu*-specific errors, such as I/O errors. *KeyError* is raised for general mapping errors like specifying an incorrect key.

`dbm.gnu.`**open**(*filename*, *flag='r'*, *mode=0o666*, /)

Open a GDBM database and return a `gdbm` object.

Paramètres

— **filename** (*path-like object*) -- The database file to open.

— **flag** (`str`) --

— `'r'` (default) : Open existing database for reading only.

— `'w'` : Open existing database for reading and writing.

— `'c'` : Open database for reading and writing, creating it if it doesn't exist.

— `'n'` : Always create a new, empty database, open for reading and writing.

The following additional characters may be appended to control how the database is opened :

— `'f'` : Open the database in fast mode. Writes to the database will not be synchronized.

— `'s'` : Synchronized mode. Changes to the database will be written immediately to the file.

— `'u'` : Do not lock database.

Not all flags are valid for all versions of GDBM. See the *open_flags* member for a list of supported flag characters.

— **mode** (`int`) -- The Unix file access mode of the file (default : octal `0o666`), used only when the database has to be created.

Lève

error -- If an invalid *flag* argument is passed.

Modifié dans la version 3.11 : *filename* accepts a *path-like object*.

`dbm.gnu.`**open_flags**

A string of characters the *flag* parameter of *open()* supports.

`gdbm` objects behave similar to *mappings*, but `items()` and `values()` methods are not supported. The following methods are also provided :

`gdbm.`**firstkey**()

It's possible to loop over every key in the database using this method and the *nextkey()* method. The traversal is ordered by GDBM's internal hash values, and won't be sorted by the key values. This method returns the starting key.

`gdbm.`**nextkey**(*key*)

Returns the key that follows *key* in the traversal. The following code prints every key in the database `db`, without having to create a list in memory that contains them all :

```
k = db.firstkey()
while k is not None:
    print(k)
    k = db.nextkey(k)
```

gdbm.**reorganize**()

If you have carried out a lot of deletions and would like to shrink the space used by the GDBM file, this routine will reorganize the database. gdbm objects will not shorten the length of a database file except by using this reorganization ; otherwise, deleted file space will be kept and reused as new (key, value) pairs are added.

gdbm.**sync**()

When the database has been opened in fast mode, this method forces any unwritten data to be written to the disk.

gdbm.**close**()

Close the GDBM database.

gdbm.**clear**()

Remove all items from the GDBM database.

Ajouté dans la version 3.13.

12.5.3 dbm.ndbm --- New Database Manager

Source code : Lib/dbm/ndbm.py

The *dbm.ndbm* module provides an interface to the NDBM (New Database Manager) library. This module can be used with the "classic" NDBM interface or the GDBM compatibility interface.

> **Note**
>
> The file formats created by *dbm.gnu* and *dbm.ndbm* are incompatible and can not be used interchangeably.

> **Avertissement**
>
> The NDBM library shipped as part of macOS has an undocumented limitation on the size of values, which can result in corrupted database files when storing values larger than this limit. Reading such corrupted files can result in a hard crash (segmentation fault).

Availability : not Android, not iOS, not WASI.

This module is not supported on *mobile platforms* or *WebAssembly platforms*.

exception dbm.ndbm.**error**

Raised on *dbm.ndbm*-specific errors, such as I/O errors. *KeyError* is raised for general mapping errors like specifying an incorrect key.

dbm.ndbm.**library**

Name of the NDBM implementation library used.

dbm.ndbm.**open**(*filename*, *flag='r'*, *mode=0o666*, /)

Open an NDBM database and return an ndbm object.

Paramètres

— **filename** (*path-like object*) -- The basename of the database file (without the .dir or .pag extensions).

- **flag** (str) --
  - 'r' (default) : Open existing database for reading only.
  - 'w' : Open existing database for reading and writing.
  - 'c' : Open database for reading and writing, creating it if it doesn't exist.
  - 'n' : Always create a new, empty database, open for reading and writing.
- **mode** (int) -- The Unix file access mode of the file (default : octal 0o666), used only when the database has to be created.

ndbm objects behave similar to *mappings*, but items() and values() methods are not supported. The following methods are also provided :

Modifié dans la version 3.11 : Accepts *path-like object* for filename.

ndbm.**close**()

Close the NDBM database.

ndbm.**clear**()

Remove all items from the NDBM database.

Ajouté dans la version 3.13.

12.5.4 dbm.dumb --- Portable DBM implementation

Source code : Lib/dbm/dumb.py

> **Note**
>
> The dbm.dumb module is intended as a last resort fallback for the dbm module when a more robust module is not available. The dbm.dumb module is not written for speed and is not nearly as heavily used as the other database modules.

The dbm.dumb module provides a persistent dict-like interface which is written entirely in Python. Unlike other dbm backends, such as dbm.gnu, no external library is required.

The dbm.dumb module defines the following :

exception dbm.dumb.**error**

Raised on dbm.dumb-specific errors, such as I/O errors. KeyError is raised for general mapping errors like specifying an incorrect key.

dbm.dumb.**open**(*filename*, *flag='c'*, *mode=0o666*)

Open a dbm.dumb database. The returned database object behaves similar to a *mapping*, in addition to providing sync() and close() methods.

Paramètres

- **filename** -- The basename of the database file (without extensions). A new database creates the following files :
  - *filename*.dat
  - *filename*.dir
- **flag** (str) --
  - 'r' : Open existing database for reading only.
  - 'w' : Open existing database for reading and writing.
  - 'c' (default) : Open database for reading and writing, creating it if it doesn't exist.
  - 'n' : Always create a new, empty database, open for reading and writing.
- **mode** (int) -- The Unix file access mode of the file (default : octal 0o666), used only when the database has to be created.

> **Avertissement**
>
> It is possible to crash the Python interpreter when loading a database with a sufficiently large/complex entry due to stack depth limitations in Python's AST compiler.

Modifié dans la version 3.5 : `open()` always creates a new database when *flag* is `'n'`.

Modifié dans la version 3.8 : A database opened read-only if *flag* is `'r'`. A database is not created if it does not exist if *flag* is `'r'` or `'w'`.

Modifié dans la version 3.11 : *filename* accepts a *path-like object*.

In addition to the methods provided by the `collections.abc.MutableMapping` class, the following methods are provided :

`dumbdbm.sync()`

Synchronize the on-disk directory and data files. This method is called by the `shelve.Shelf.sync()` method.

`dumbdbm.close()`

Close the database.

12.6 `sqlite3` --- DB-API 2.0 interface for SQLite databases

Code source : Lib/sqlite3/ SQLite est une bibliothèque C qui fournit une base de données légère sur disque ne nécessitant pas de processus serveur et qui utilise une variante (non standard) du langage de requête SQL pour accéder aux données. Certaines applications peuvent utiliser SQLite pour le stockage de données internes. Il est également possible de créer une application prototype utilisant SQLite, puis de modifier le code pour utiliser une base de données plus robuste telle que PostgreSQL ou Oracle.

The `sqlite3` module was written by Gerhard Häring. It provides an SQL interface compliant with the DB-API 2.0 specification described by **PEP 249**, and requires SQLite 3.15.2 or newer.

Ce document inclus 4 sections principales :

- *tutoriel sqlite3* explique comment utiliser le module `sqlite3`.
- *référence sqlite3* décrit les classes et les fonctions que ce module définit.
- *guide sqlite3* détaille comment gérer des tâches spécifiques.
- *explications sqlite3* propose un contexte détaillé du contrôle de transaction.

> **Voir aussi**
>
> **https://www.sqlite.org**
> Dans la page Web de SQLite, la documentation décrit la syntaxe et les types de données disponibles qui sont pris en charge par cette variante SQL.
>
> **https://www.w3schools.com/sql/**
> Tutoriel, référence et exemples pour apprendre la syntaxe SQL.
>
> **PEP 249 — Spécifications de l'API 2.0 pour la base de données**
> PEP écrite par Marc-André Lemburg.

12.6.1 Tutoriel

Dans ce tutoriel, vous allez créer une base de données des films des Monty Python en utilisant les fonctionnalités de base de `sqlite3`. Cela nécessite une compréhension élémentaire des concepts des bases de données, notamment les curseurs et les transactions.

Tout d'abord, nous devons créer une nouvelle base de données et ouvrir une connexion à la base de données pour permettre à `sqlite3` de travailler avec elle. Appelez `sqlite3.connect()` pour créer une connexion à la base de données `tutorial.db` dans le répertoire de travail actuel, en la créant implicitement si elle n'existe pas :

```
import sqlite3
con = sqlite3.connect("tutorial.db")
```

L'objet *Connection* renvoyé — `con` — représente la connexion à la base de données sur disque.

Afin d'exécuter les instructions SQL et de récupérer les résultats des requêtes SQL, nous devrons utiliser un curseur de base de données. Appelez *con.cursor()* pour créer la *Cursor* :

```
cur = con.cursor()
```

Maintenant que nous avons une connexion à la base de données et un curseur, nous pouvons créer une table `movie` avec des colonnes pour le titre, l'année de sortie et la note de la critique. Pour plus de simplicité, nous pouvons simplement utiliser les noms des colonnes dans la déclaration de la table — grâce à la fonctionnalité de typage flexible de SQLite, spécifier les types de données est facultatif. Exécutez l'instruction `CREATE TABLE` en appelant *cur.execute(…)* :

```
cur.execute("CREATE TABLE movie(title, year, score)")
```

Nous pouvons vérifier que la nouvelle table a été créée en interrogeant la table `sqlite_master` intégrée à SQLite, qui devrait maintenant contenir une entrée pour la définition de la table `movie` (voir le schéma Table pour plus de détails). Exécutez cette requête en appelant *cur.execute(…)*, affectez le résultat à `res`, et appelez *res.fetchone()* pour récupérer la ligne résultante :

```
>>> res = cur.execute("SELECT name FROM sqlite_master")
>>> res.fetchone()
('movie',)
```

We can see that the table has been created, as the query returns a *tuple* containing the table's name. If we query `sqlite_master` for a non-existent table `spam`, `res.fetchone()` will return `None` :

```
>>> res = cur.execute("SELECT name FROM sqlite_master WHERE name='spam'")
>>> res.fetchone() is None
True
```

Maintenant, ajoutez deux lignes de données en tant que littéraux SQL en exécutant une instruction `INSERT`, une fois encore en appelant *cur.execute(…)* :

```
cur.execute("""
    INSERT INTO movie VALUES
        ('Monty Python and the Holy Grail', 1975, 8.2),
        ('And Now for Something Completely Different', 1971, 7.5)
""")
```

L'instruction `INSERT` ouvre implicitement une transaction, qui doit être validée avant que les modifications ne soient enregistrées dans la base de données (voir *contrôle des transactions SQL* pour plus de détails). Appelez *con.commit()* sur l'objet de connexion pour valider la transaction :

```
con.commit()
```

Nous pouvons vérifier que les données ont été insérées correctement en exécutant une requête `SELECT`. Utilisez la désormais familière *cur.execute(…)* pour affecter le résultat à `res`, et appelez *res.fetchall()* pour retourner toutes les lignes résultantes :

```
>>> res = cur.execute("SELECT score FROM movie")
>>> res.fetchall()
[(8.2,), (7.5,)]
```

Le résultat est une *liste* de deux `tuple`s, une par ligne, chacun contenant la valeur `score` de cette ligne.

Maintenant, insérez trois lignes supplémentaires en appelant *cur.executemany(…)* :

```
data = [
    ("Monty Python Live at the Hollywood Bowl", 1982, 7.9),
    ("Monty Python's The Meaning of Life", 1983, 7.5),
    ("Monty Python's Life of Brian", 1979, 8.0),
]
cur.executemany("INSERT INTO movie VALUES(?, ?, ?)", data)
con.commit()  # Remember to commit the transaction after executing INSERT.
```

Remarquez que les marqueurs `?` sont utilisés pour lier les `data` à la requête. Utilisez toujours les marqueurs au lieu d'expressions formatées pour lier les valeurs Python aux instructions SQL, afin d'éviter les injections SQL (voir *placeholder SQL* pour plus de détails).

Nous pouvons vérifier que les nouvelles lignes ont été insérées en exécutant une requête `SELECT`, cette fois-ci en itérant sur les résultats de la requête :

```
>>> for row in cur.execute("SELECT year, title FROM movie ORDER BY year"):
...     print(row)
(1971, 'And Now for Something Completely Different')
(1975, 'Monty Python and the Holy Grail')
(1979, "Monty Python's Life of Brian")
(1982, 'Monty Python Live at the Hollywood Bowl')
(1983, "Monty Python's The Meaning of Life")
```

Chaque ligne est un *tuple* de deux éléments `(année, titre)`, correspondant aux colonnes sélectionnées dans la requête.

Enfin, vérifiez que la base de données a été écrite sur le disque en appelant *`con.close()`* pour fermer la connexion existante, en ouvrir une nouvelle, créer un nouveau curseur, puis interroger la base de données :

```
>>> con.close()
>>> new_con = sqlite3.connect("tutorial.db")
>>> new_cur = new_con.cursor()
>>> res = new_cur.execute("SELECT title, year FROM movie ORDER BY score DESC")
>>> title, year = res.fetchone()
>>> print(f'The highest scoring Monty Python movie is {title!r}, released in {year}
↪')
The highest scoring Monty Python movie is 'Monty Python and the Holy Grail',␣
↪released in 1975
>>> new_con.close()
```

Vous avez maintenant créé une base de données SQLite à l'aide du module `sqlite3`, inséré des données et récupéré des valeurs de plusieurs façons.

Voir aussi

- *guide sqlite3* pour une lecture plus poussée :
  - *placeholders sqlite3*
  - *adaptateurs sqlite3*
  - *convertisseurs sqlite3*
  - *gestionnaire de contexte de connexion sqlite3*
  - *guide de fabrique de ligne sqlite3*
- *explications sqlite3* pour un contexte détaillé du contrôle de transaction.

12.6.2 Références

Fonctions du module

`sqlite3.connect`(*database*, *timeout=5.0*, *detect_types=0*, *isolation_level='DEFERRED'*, *check_same_thread=True*, *factory=sqlite3.Connection*, *cached_statements=128*, *uri=False*, ***, *autocommit=sqlite3.LEGACY_TRANSACTION_CONTROL*)

Ouvrez une connexion à une base de données SQLite.

Paramètres

- **database** (*path-like object*) -- Le chemin d'accès au fichier de la base de données à ouvrir. Passez `":memory :"` pour créer une base de données SQLite existant uniquement en mémoire, et ouvrir une connexion à celle-ci.
- **timeout** (`float`) -- Le temps (en secondes) que la connexion doit attendre avant de lever une `OperationalError`, si la table est verrouillée. Si une autre connexion ouvre une transaction pour modifier la table, celle-ci sera verrouillée jusqu'à ce que cette transaction soit validée. Par défaut, cinq secondes.
- **detect_types** (`int`) -- Contrôle si et comment les types de données non *nativement pris en charge par SQLite* sont recherchés pour être convertis en types Python, en utilisant les convertisseurs enregistrés avec `register_converter()`. Définissez-le à n'importe quelle combinaison (en utilisant `|`, opérateurs bit-à-bit OR) de `PARSE_DECLTYPES` et `PARSE_COLNAMES` pour activer ceci. Les noms de colonnes ont la priorité sur les types déclarés si les deux drapeaux sont activés. Les types ne peuvent pas être détectés pour les champs générés (par exemple `max(data)`), même si le paramètre *detect_types* est activé ; `str` sera retourné à la place. Par défaut (`0`), la détection des types est désactivée.
- **isolation_level** (`str | None`) -- Control legacy transaction handling behaviour. See `Connection.isolation_level` and *Transaction control via the isolation_level attribute* for more information. Can be `"DEFERRED"` (default), `"EXCLUSIVE"` or `"IMMEDIATE"` ; or `None` to disable opening transactions implicitly. Has no effect unless `Connection.autocommit` is set to `LEGACY_TRANSACTION_CONTROL` (the default).
- **check_same_thread** (`bool`) -- Si `True` (par défaut), `ProgrammingError` sera levée si la connexion à la base de données est utilisée par un thread autre que celui qui l'a créée. Si `False`, la connexion peut être utilisée par plusieurs threads ; les opérations d'écriture devront peut-être être sérialisées par l'utilisateur pour éviter la corruption des données. Voir `sécurité des threads` pour plus d'informations.
- **factory** (`Connection`) -- Une sous-classe personnalisée de `Connection` pour créer la connexion, si ce n'est pas la classe par défaut `Connection`.
- **cached_statements** (`int`) -- Le nombre d'instructions que `sqlite3` doit mettre en cache en interne pour cette connexion, afin d'éviter les surcharges d'analyse. Par défaut, 128 instructions.
- **uri** (`bool`) -- Si elle a pour valeur `True`, la base de données est interprétée comme un URI (Uniform Resource Identifier) avec un chemin d'accès au fichier et une chaîne de requête facultative. La partie schéma *doit* être `"file:"`, et le chemin peut être relatif ou absolu. La chaîne d'interrogation permet de passer des paramètres à SQLite, ce qui permet d'activer diverses *astuces d'URI sqlite3*.
- **autocommit** (`bool`) -- Control **PEP 249** transaction handling behaviour. See `Connection.autocommit` and *Transaction control via the autocommit attribute* for more information. *autocommit* currently defaults to `LEGACY_TRANSACTION_CONTROL`. The default will change to `False` in a future Python release.

Type renvoyé

Connection

Raises an *auditing event* `sqlite3.connect` with argument `database`.

Raises an *auditing event* `sqlite3.connect/handle` with argument `connection_handle`.

Modifié dans la version 3.4 : Added the *uri* parameter.

Modifié dans la version 3.7 : *database* peut maintenant aussi être un *objet de type chemin*, et pas seulement une chaîne de caractères.

Modifié dans la version 3.10 : Added the `sqlite3.connect/handle` auditing event.

Modifié dans la version 3.12 : Added the *autocommit* parameter.

Modifié dans la version 3.13 : Positional use of the parameters *timeout*, *detect_types*, *isolation_level*, *check_same_thread*, *factory*, *cached_statements*, and *uri* is deprecated. They will become keyword-only parameters in Python 3.15.

`sqlite3.`**`complete_statement`**(*statement*)

Renvoie `True` si la déclaration de la chaîne semble contenir une ou plusieurs déclarations SQL complètes. Aucune vérification syntaxique ou analyse syntaxique d'aucune sorte n'est effectuée, si ce n'est la vérification qu'il n'y a pas de chaîne littérale non fermée et que l'instruction se termine par un point-virgule.

Exemple :

```
>>> sqlite3.complete_statement("SELECT foo FROM bar;")
True
>>> sqlite3.complete_statement("SELECT foo")
False
```

Cette fonction peut être utile pendant la saisie en ligne de commande pour déterminer si le texte saisi semble former une instruction SQL complète, ou si une saisie supplémentaire est nécessaire avant d'appeler `execute()`.

See `runsource()` in Lib/sqlite3/__main__.py for real-world use.

`sqlite3.`**`enable_callback_tracebacks`**(*flag*, /)

Activer ou désactiver les traces des fonctions de rappel. Par défaut, vous n'obtiendrez pas de traces de pile d'appels dans les fonctions définies par l'utilisateur, les agrégats, les convertisseurs, les fonctions de rappel des mécanismes d'autorisation, etc. Si vous voulez les déboguer, vous pouvez appeler cette fonction avec *flag* à `True`. Ensuite, vous obtiendrez les traces des fonctions de rappel sur `sys.stderr`. Utilisez `False` pour désactiver à nouveau cette fonctionnalité.

> **Note**
>
> Errors in user-defined function callbacks are logged as unraisable exceptions. Use an `unraisable hook handler` for introspection of the failed callback.

`sqlite3.`**`register_adapter`**(*type*, *adapter*, /)

Enregistre un *adaptateur callable* pour adapter le type Python *type* en un type SQLite. L'adaptateur est appelé avec un objet Python de type *type* comme seul argument, et doit retourner une valeur d'un *type que SQLite comprend nativement*.

`sqlite3.`**`register_converter`**(*typename*, *converter*, /)

Enregistrer le *convertisseur callable* pour convertir les objets SQLite de type *typename* en un objet Python d'un type spécifique. Le convertisseur est invoqué pour toutes les valeurs SQLite de type *typename* ; on lui passe un objet `bytes` et il doit retourner un objet du type Python désiré. Consultez le paramètre *detect_types* de `connect()` pour plus d'informations sur le fonctionnement de la détection de type.

Remarque : *typename* et le nom du type dans votre requête sont comparés sans tenir compte de la casse.

Fonctions et constantes du module

`sqlite3.`**`LEGACY_TRANSACTION_CONTROL`**

Set `autocommit` to this constant to select old style (pre-Python 3.12) transaction control behaviour. See *Transaction control via the isolation_level attribute* for more information.

`sqlite3.`**`PARSE_COLNAMES`**

Pass this flag value to the *detect_types* parameter of `connect()` to look up a converter function by using the type name, parsed from the query column name, as the converter dictionary key. The type name must be wrapped in square brackets (`[]`).

```
SELECT p as "p [point]" FROM test;  ! will look up converter "point"
```

This flag may be combined with *PARSE_DECLTYPES* using the | (bitwise or) operator.

sqlite3.**PARSE_DECLTYPES**

Pass this flag value to the *detect_types* parameter of *connect()* to look up a converter function using the declared types for each column. The types are declared when the database table is created. sqlite3 will look up a converter function using the first word of the declared type as the converter dictionary key. For example :

```
CREATE TABLE test(
   i integer primary key,  ! will look up a converter named "integer"
   p point,                ! will look up a converter named "point"
   n number(10)            ! will look up a converter named "number"
 )
```

This flag may be combined with *PARSE_COLNAMES* using the | (bitwise or) operator.

sqlite3.**SQLITE_OK**

sqlite3.**SQLITE_DENY**

sqlite3.**SQLITE_IGNORE**

Flags that should be returned by the *authorizer_callback* *callable* passed to *Connection.set_authorizer()*, to indicate whether :

— Access is allowed (SQLITE_OK),
— The SQL statement should be aborted with an error (SQLITE_DENY)
— The column should be treated as a NULL value (SQLITE_IGNORE)

sqlite3.**apilevel**

String constant stating the supported DB-API level. Required by the DB-API. Hard-coded to "2.0".

sqlite3.**paramstyle**

String constant stating the type of parameter marker formatting expected by the sqlite3 module. Required by the DB-API. Hard-coded to "qmark".

> **Note**
>
> The named DB-API parameter style is also supported.

sqlite3.**sqlite_version**

Version number of the runtime SQLite library as a *string*.

sqlite3.**sqlite_version_info**

Version number of the runtime SQLite library as a *tuple* of *integers*.

sqlite3.**threadsafety**

Integer constant required by the DB-API 2.0, stating the level of thread safety the sqlite3 module supports. This attribute is set based on the default threading mode the underlying SQLite library is compiled with. The SQLite threading modes are :

1. **Single-thread** : In this mode, all mutexes are disabled and SQLite is unsafe to use in more than a single thread at once.
2. **Multi-thread** : In this mode, SQLite can be safely used by multiple threads provided that no single database connection is used simultaneously in two or more threads.
3. **Serialized** : In serialized mode, SQLite can be safely used by multiple threads with no restriction.

The mappings from SQLite threading modes to DB-API 2.0 threadsafety levels are as follows :

SQLite threading mode	threadsafety	SQLITE_THREADSAFE	DB-API 2.0 meaning
single-thread	0	0	Threads may not share the module
multi-thread	1	2	Threads may share the module, but not connections
serialized	3	1	Threads may share the module, connections and cursors

Modifié dans la version 3.11 : Set *threadsafety* dynamically instead of hard-coding it to `1`.

`sqlite3.`**`SQLITE_DBCONFIG_DEFENSIVE`**

`sqlite3.`**`SQLITE_DBCONFIG_DQS_DDL`**

`sqlite3.`**`SQLITE_DBCONFIG_DQS_DML`**

`sqlite3.`**`SQLITE_DBCONFIG_ENABLE_FKEY`**

`sqlite3.`**`SQLITE_DBCONFIG_ENABLE_FTS3_TOKENIZER`**

`sqlite3.`**`SQLITE_DBCONFIG_ENABLE_LOAD_EXTENSION`**

`sqlite3.`**`SQLITE_DBCONFIG_ENABLE_QPSG`**

`sqlite3.`**`SQLITE_DBCONFIG_ENABLE_TRIGGER`**

`sqlite3.`**`SQLITE_DBCONFIG_ENABLE_VIEW`**

`sqlite3.`**`SQLITE_DBCONFIG_LEGACY_ALTER_TABLE`**

`sqlite3.`**`SQLITE_DBCONFIG_LEGACY_FILE_FORMAT`**

`sqlite3.`**`SQLITE_DBCONFIG_NO_CKPT_ON_CLOSE`**

`sqlite3.`**`SQLITE_DBCONFIG_RESET_DATABASE`**

`sqlite3.`**`SQLITE_DBCONFIG_TRIGGER_EQP`**

`sqlite3.`**`SQLITE_DBCONFIG_TRUSTED_SCHEMA`**

`sqlite3.`**`SQLITE_DBCONFIG_WRITABLE_SCHEMA`**

These constants are used for the `Connection.setconfig()` and `getconfig()` methods.

The availability of these constants varies depending on the version of SQLite Python was compiled with.

Ajouté dans la version 3.12.

Voir aussi

https://www.sqlite.org/c3ref/c_dbconfig_defensive.html
SQLite docs : Database Connection Configuration Options

Deprecated since version 3.12, removed in version 3.14 : The `version` and `version_info` constants.

Connection objects

class `sqlite3.`**`Connection`**

Each open SQLite database is represented by a `Connection` object, which is created using `sqlite3.connect()`. Their main purpose is creating `Cursor` objects, and *Transaction control*.

Voir aussi

— *How to use connection shortcut methods*
— *gestionnaire de contexte de connexion sqlite3*

Modifié dans la version 3.13 : A `ResourceWarning` is emitted if `close()` is not called before a `Connection` object is deleted.

An SQLite database connection has the following attributes and methods :

cursor(*factory=Cursor*)

Create and return a `Cursor` object. The cursor method accepts a single optional parameter *factory*. If supplied, this must be a *callable* returning an instance of `Cursor` or its subclasses.

blobopen(*table*, *column*, *row*, /, *, *readonly=False*, *name='main'*)

Open a `Blob` handle to an existing BLOB (Binary Large OBject).

Paramètres

- **table** (`str`) -- The name of the table where the blob is located.
- **column** (`str`) -- The name of the column where the blob is located.
- **row** (`str`) -- The name of the row where the blob is located.
- **readonly** (`bool`) -- Set to `True` if the blob should be opened without write permissions. Defaults to `False`.
- **name** (`str`) -- The name of the database where the blob is located. Defaults to `"main"`.

Lève

OperationalError -- When trying to open a blob in a `WITHOUT ROWID` table.

Type renvoyé

Blob

> **Note**
>
> The blob size cannot be changed using the `Blob` class. Use the SQL function `zeroblob` to create a blob with a fixed size.

Ajouté dans la version 3.11.

commit()

Commit any pending transaction to the database. If `autocommit` is `True`, or there is no open transaction, this method does nothing. If `autocommit` is `False`, a new transaction is implicitly opened if a pending transaction was committed by this method.

rollback()

Roll back to the start of any pending transaction. If `autocommit` is `True`, or there is no open transaction, this method does nothing. If `autocommit` is `False`, a new transaction is implicitly opened if a pending transaction was rolled back by this method.

close()

Close the database connection. If `autocommit` is `False`, any pending transaction is implicitly rolled back. If `autocommit` is `True` or `LEGACY_TRANSACTION_CONTROL`, no implicit transaction control is executed. Make sure to `commit()` before closing to avoid losing pending changes.

execute(*sql*, *parameters=()*, /)

Create a new `Cursor` object and call `execute()` on it with the given *sql* and *parameters*. Return the new cursor object.

executemany(*sql*, *parameters*, /)

Create a new `Cursor` object and call `executemany()` on it with the given *sql* and *parameters*. Return the new cursor object.

executescript(*sql_script*, /)

Create a new `Cursor` object and call `executescript()` on it with the given *sql_script*. Return the new cursor object.

create_function(*name*, *narg*, *func*, *, *deterministic=False*)

Create or remove a user-defined SQL function.

Paramètres

- **name** (`str`) -- The name of the SQL function.
- **narg** (`int`) -- The number of arguments the SQL function can accept. If `-1`, it may take any number of arguments.

— **func** (*callback* | None) -- A *callable* that is called when the SQL function is invoked. The callable must return *a type natively supported by SQLite*. Set to `None` to remove an existing SQL function.
— **deterministic** (`bool`) -- If `True`, the created SQL function is marked as deterministic, which allows SQLite to perform additional optimizations.

Modifié dans la version 3.8 : Added the *deterministic* parameter.

Exemple :

```
>>> import hashlib
>>> def md5sum(t):
...     return hashlib.md5(t).hexdigest()
>>> con = sqlite3.connect(":memory:")
>>> con.create_function("md5", 1, md5sum)
>>> for row in con.execute("SELECT md5(?)", (b"foo",)):
...     print(row)
('acbd18db4cc2f85cedef654fccc4a4d8',)
>>> con.close()
```

Modifié dans la version 3.13 : Passing *name*, *narg*, and *func* as keyword arguments is deprecated. These parameters will become positional-only in Python 3.15.

create_aggregate(*name*, *n_arg*, *aggregate_class*)

Create or remove a user-defined SQL aggregate function.

Paramètres

— **name** (`str`) -- The name of the SQL aggregate function.
— **n_arg** (`int`) -- The number of arguments the SQL aggregate function can accept. If `-1`, it may take any number of arguments.
— **aggregate_class** (*class* | None) -- A class must implement the following methods :
  — `step()` : Add a row to the aggregate.
  — `finalize()` : Return the final result of the aggregate as *a type natively supported by SQLite*.

  The number of arguments that the `step()` method must accept is controlled by *n_arg*.

  Set to `None` to remove an existing SQL aggregate function.

Exemple :

```
class MySum:
    def __init__(self):
        self.count = 0

    def step(self, value):
        self.count += value

    def finalize(self):
        return self.count

con = sqlite3.connect(":memory:")
con.create_aggregate("mysum", 1, MySum)
cur = con.execute("CREATE TABLE test(i)")
cur.execute("INSERT INTO test(i) VALUES(1)")
cur.execute("INSERT INTO test(i) VALUES(2)")
cur.execute("SELECT mysum(i) FROM test")
print(cur.fetchone()[0])

con.close()
```

Modifié dans la version 3.13 : Passing *name*, *n_arg*, and *aggregate_class* as keyword arguments is deprecated. These parameters will become positional-only in Python 3.15.

create_window_function(*name*, *num_params*, *aggregate_class*, /)

Create or remove a user-defined aggregate window function.

Paramètres

- **name** (str) -- The name of the SQL aggregate window function to create or remove.
- **num_params** (int) -- The number of arguments the SQL aggregate window function can accept. If -1, it may take any number of arguments.
- **aggregate_class** (*class* | None) -- A class that must implement the following methods :
  - step() : Add a row to the current window.
  - value() : Return the current value of the aggregate.
  - inverse() : Remove a row from the current window.
  - finalize() : Return the final result of the aggregate as *a type natively supported by SQLite*.

  The number of arguments that the step() and value() methods must accept is controlled by *num_params*.

  Set to None to remove an existing SQL aggregate window function.

Lève

NotSupportedError -- If used with a version of SQLite older than 3.25.0, which does not support aggregate window functions.

Ajouté dans la version 3.11.

Exemple :

```
# Example taken from https://www.sqlite.org/windowfunctions.html#udfwinfunc
class WindowSumInt:
    def __init__(self):
        self.count = 0

    def step(self, value):
        """Add a row to the current window."""
        self.count += value

    def value(self):
        """Return the current value of the aggregate."""
        return self.count

    def inverse(self, value):
        """Remove a row from the current window."""
        self.count -= value

    def finalize(self):
        """Return the final value of the aggregate.

        Any clean-up actions should be placed here.
        """
        return self.count


con = sqlite3.connect(":memory:")
cur = con.execute("CREATE TABLE test(x, y)")
values = [
    ("a", 4),
    ("b", 5),
    ("c", 3),
    ("d", 8),
    ("e", 1),
]
```

(suite sur la page suivante)

(suite de la page précédente)

```
cur.executemany("INSERT INTO test VALUES(?, ?)", values)
con.create_window_function("sumint", 1, WindowSumInt)
cur.execute("""
    SELECT x, sumint(y) OVER (
        ORDER BY x ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING
    ) AS sum_y
    FROM test ORDER BY x
""")
print(cur.fetchall())
con.close()
```

create_collation(*name*, *callable*, /)

Create a collation named *name* using the collating function *callable*. *callable* is passed two `string` arguments, and it should return an `integer` :
— `1` if the first is ordered higher than the second
— `-1` if the first is ordered lower than the second
— `0` if they are ordered equal
The following example shows a reverse sorting collation :

```
def collate_reverse(string1, string2):
    if string1 == string2:
        return 0
    elif string1 < string2:
        return 1
    else:
        return -1

con = sqlite3.connect(":memory:")
con.create_collation("reverse", collate_reverse)

cur = con.execute("CREATE TABLE test(x)")
cur.executemany("INSERT INTO test(x) VALUES(?)", [("a",), ("b",)])
cur.execute("SELECT x FROM test ORDER BY x COLLATE reverse")
for row in cur:
    print(row)
con.close()
```

Remove a collation function by setting *callable* to `None`.

Modifié dans la version 3.11 : The collation name can contain any Unicode character. Earlier, only ASCII characters were allowed.

interrupt()

Call this method from a different thread to abort any queries that might be executing on the connection. Aborted queries will raise an `OperationalError`.

set_authorizer(*authorizer_callback*)

Register *callable* *authorizer_callback* to be invoked for each attempt to access a column of a table in the database. The callback should return one of `SQLITE_OK`, `SQLITE_DENY`, or `SQLITE_IGNORE` to signal how access to the column should be handled by the underlying SQLite library.

The first argument to the callback signifies what kind of operation is to be authorized. The second and third argument will be arguments or `None` depending on the first argument. The 4th argument is the name of the database ("main", "temp", etc.) if applicable. The 5th argument is the name of the innermost trigger or view that is responsible for the access attempt or `None` if this access attempt is directly from input SQL code.

Please consult the SQLite documentation about the possible values for the first argument and the meaning of the second and third argument depending on the first one. All necessary constants are available in the

`sqlite3` module.

Passing `None` as *authorizer_callback* will disable the authorizer.

Modifié dans la version 3.11 : Added support for disabling the authorizer using `None`.

Modifié dans la version 3.13 : Passing *authorizer_callback* as a keyword argument is deprecated. The parameter will become positional-only in Python 3.15.

set_progress_handler(*progress_handler*, *n*)

Register *callable* *progress_handler* to be invoked for every *n* instructions of the SQLite virtual machine. This is useful if you want to get called from SQLite during long-running operations, for example to update a GUI.

If you want to clear any previously installed progress handler, call the method with `None` for *progress_handler*.

Returning a non-zero value from the handler function will terminate the currently executing query and cause it to raise a `DatabaseError` exception.

Modifié dans la version 3.13 : Passing *progress_handler* as a keyword argument is deprecated. The parameter will become positional-only in Python 3.15.

set_trace_callback(*trace_callback*)

Register *callable* *trace_callback* to be invoked for each SQL statement that is actually executed by the SQLite backend.

The only argument passed to the callback is the statement (as `str`) that is being executed. The return value of the callback is ignored. Note that the backend does not only run statements passed to the `Cursor.execute()` methods. Other sources include the *transaction management* of the `sqlite3` module and the execution of triggers defined in the current database.

Passing `None` as *trace_callback* will disable the trace callback.

> **Note**
>
> Exceptions raised in the trace callback are not propagated. As a development and debugging aid, use `enable_callback_tracebacks()` to enable printing tracebacks from exceptions raised in the trace callback.

Ajouté dans la version 3.3.

Modifié dans la version 3.13 : Passing *trace_callback* as a keyword argument is deprecated. The parameter will become positional-only in Python 3.15.

enable_load_extension(*enabled*, /)

Enable the SQLite engine to load SQLite extensions from shared libraries if *enabled* is `True`; else, disallow loading SQLite extensions. SQLite extensions can define new functions, aggregates or whole new virtual table implementations. One well-known extension is the fulltext-search extension distributed with SQLite.

> **Note**
>
> The `sqlite3` module is not built with loadable extension support by default, because some platforms (notably macOS) have SQLite libraries which are compiled without this feature. To get loadable extension support, you must pass the `--enable-loadable-sqlite-extensions` option to **configure**.

Raises an *auditing event* `sqlite3.enable_load_extension` with arguments `connection`, `enabled`.

Ajouté dans la version 3.2.

Modifié dans la version 3.10 : Added the `sqlite3.enable_load_extension` auditing event.

```
con.enable_load_extension(True)

# Load the fulltext search extension
con.execute("select load_extension('./fts3.so')")

# alternatively you can load the extension using an API call:
# con.load_extension("./fts3.so")

# disable extension loading again
con.enable_load_extension(False)

# example from SQLite wiki
con.execute("CREATE VIRTUAL TABLE recipe USING fts3(name, ingredients)")
con.executescript("""
    INSERT INTO recipe (name, ingredients) VALUES('broccoli stew',
↪'broccoli peppers cheese tomatoes');
    INSERT INTO recipe (name, ingredients) VALUES('pumpkin stew', 'pumpkin␣
↪onions garlic celery');
    INSERT INTO recipe (name, ingredients) VALUES('broccoli pie',
↪'broccoli cheese onions flour');
    INSERT INTO recipe (name, ingredients) VALUES('pumpkin pie', 'pumpkin␣
↪sugar flour butter');
    """)
for row in con.execute("SELECT rowid, name, ingredients FROM recipe WHERE␣
↪name MATCH 'pie'"):
    print(row)
```

load_extension(*path*, /, *, *entrypoint=None*)

Load an SQLite extension from a shared library. Enable extension loading with `enable_load_extension()` before calling this method.

Paramètres

- **path** (`str`) -- The path to the SQLite extension.
- **entrypoint** (`str` | *None*) -- Entry point name. If `None` (the default), SQLite will come up with an entry point name of its own ; see the SQLite docs Loading an Extension for details.

Raises an *auditing event* `sqlite3.load_extension` with arguments `connection`, `path`.

Ajouté dans la version 3.2.

Modifié dans la version 3.10 : Added the `sqlite3.load_extension` auditing event.

Modifié dans la version 3.12 : Added the *entrypoint* parameter.

iterdump(*, *filter=None*)

Return an *iterator* to dump the database as SQL source code. Useful when saving an in-memory database for later restoration. Similar to the `.dump` command in the **sqlite3** shell.

Paramètres

filter (`str` | *None*) -- An optional `LIKE` pattern for database objects to dump, e.g. `prefix_%`. If `None` (the default), all database objects will be included.

Exemple :

```
# Convert file example.db to SQL dump file dump.sql
con = sqlite3.connect('example.db')
with open('dump.sql', 'w') as f:
    for line in con.iterdump():
        f.write('%s\n' % line)
con.close()
```

> **Voir aussi**
>
> *How to handle non-UTF-8 text encodings*

Modifié dans la version 3.13 : Ajout du paramètre *filter*.

backup(*target*, ***, *pages=-1*, *progress=None*, *name='main'*, *sleep=0.250*)

Create a backup of an SQLite database.

Works even if the database is being accessed by other clients or concurrently by the same connection.

Paramètres

- **target** (`Connection`) -- The database connection to save the backup to.
- **pages** (`int`) -- The number of pages to copy at a time. If equal to or less than `0`, the entire database is copied in a single step. Defaults to `-1`.
- **progress** (*callback* | None) -- If set to a *callable*, it is invoked with three integer arguments for every backup iteration : the *status* of the last iteration, the *remaining* number of pages still to be copied, and the *total* number of pages. Defaults to `None`.
- **name** (`str`) -- The name of the database to back up. Either `"main"` (the default) for the main database, `"temp"` for the temporary database, or the name of a custom database as attached using the `ATTACH DATABASE` SQL statement.
- **sleep** (`float`) -- The number of seconds to sleep between successive attempts to back up remaining pages.

Example 1, copy an existing database into another :

```
def progress(status, remaining, total):
    print(f'Copied {total-remaining} of {total} pages...')

src = sqlite3.connect('example.db')
dst = sqlite3.connect('backup.db')
with dst:
    src.backup(dst, pages=1, progress=progress)
dst.close()
src.close()
```

Example 2, copy an existing database into a transient copy :

```
src = sqlite3.connect('example.db')
dst = sqlite3.connect(':memory:')
src.backup(dst)
dst.close()
src.close()
```

Ajouté dans la version 3.7.

> **Voir aussi**
>
> *How to handle non-UTF-8 text encodings*

getlimit(*category*, */*)

Get a connection runtime limit.

Paramètres

category (`int`) -- The SQLite limit category to be queried.

Type renvoyé

int

Lève

ProgrammingError -- If *category* is not recognised by the underlying SQLite library.

Example, query the maximum length of an SQL statement for `Connection` `con` (the default is 1000000000) :

```
>>> con.getlimit(sqlite3.SQLITE_LIMIT_SQL_LENGTH)
1000000000
```

Ajouté dans la version 3.11.

setlimit(*category*, *limit*, /)

Set a connection runtime limit. Attempts to increase a limit above its hard upper bound are silently truncated to the hard upper bound. Regardless of whether or not the limit was changed, the prior value of the limit is returned.

Paramètres

— **category** (`int`) -- The SQLite limit category to be set.

— **limit** (`int`) -- The value of the new limit. If negative, the current limit is unchanged.

Type renvoyé

int

Lève

ProgrammingError -- If *category* is not recognised by the underlying SQLite library.

Example, limit the number of attached databases to 1 for `Connection` `con` (the default limit is 10) :

```
>>> con.setlimit(sqlite3.SQLITE_LIMIT_ATTACHED, 1)
10
>>> con.getlimit(sqlite3.SQLITE_LIMIT_ATTACHED)
1
```

Ajouté dans la version 3.11.

getconfig(*op*, /)

Query a boolean connection configuration option.

Paramètres

op (`int`) -- A *SQLITE_DBCONFIG code*.

Type renvoyé

bool

Ajouté dans la version 3.12.

setconfig(*op*, *enable=True*, /)

Set a boolean connection configuration option.

Paramètres

— **op** (`int`) -- A *SQLITE_DBCONFIG code*.

— **enable** (`bool`) -- `True` if the configuration option should be enabled (default) ; `False` if it should be disabled.

Ajouté dans la version 3.12.

serialize(**, name='main'*)

Serialize a database into a `bytes` object. For an ordinary on-disk database file, the serialization is just a copy of the disk file. For an in-memory database or a "temp" database, the serialization is the same sequence of bytes which would be written to disk if that database were backed up to disk.

Paramètres

name (`str`) -- The database name to be serialized. Defaults to `"main"`.

Type renvoyé

bytes

> **Note**
>
> This method is only available if the underlying SQLite library has the serialize API.

Ajouté dans la version 3.11.

deserialize(*data*, /, *, *name='main'*)

Deserialize a `serialized` database into a `Connection`. This method causes the database connection to disconnect from database *name*, and reopen *name* as an in-memory database based on the serialization contained in *data*.

Paramètres

- **data** (`bytes`) -- A serialized database.
- **name** (`str`) -- The database name to deserialize into. Defaults to `"main"`.

Lève

- ***OperationalError*** -- If the database connection is currently involved in a read transaction or a backup operation.
- ***DatabaseError*** -- If *data* does not contain a valid SQLite database.
- ***OverflowError*** -- If `len(data)` is larger than `2**63 - 1`.

> **Note**
>
> This method is only available if the underlying SQLite library has the deserialize API.

Ajouté dans la version 3.11.

autocommit

This attribute controls **PEP 249**-compliant transaction behaviour. `autocommit` has three allowed values :

- `False` : Select **PEP 249**-compliant transaction behaviour, implying that `sqlite3` ensures a transaction is always open. Use `commit()` and `rollback()` to close transactions.

  This is the recommended value of `autocommit`.
- `True` : Use SQLite's autocommit mode. `commit()` and `rollback()` have no effect in this mode.
- `LEGACY_TRANSACTION_CONTROL` : Pre-Python 3.12 (non-**PEP 249**-compliant) transaction control. See `isolation_level` for more details.

  This is currently the default value of `autocommit`.

Changing `autocommit` to `False` will open a new transaction, and changing it to `True` will commit any pending transaction.

See *Transaction control via the autocommit attribute* for more details.

> **Note**
>
> The `isolation_level` attribute has no effect unless `autocommit` is `LEGACY_TRANSACTION_CONTROL`.

Ajouté dans la version 3.12.

in_transaction

This read-only attribute corresponds to the low-level SQLite autocommit mode.

`True` if a transaction is active (there are uncommitted changes), `False` otherwise.

Ajouté dans la version 3.2.

isolation_level

Controls the *legacy transaction handling mode* of `sqlite3`. If set to `None`, transactions are never implicitly opened. If set to one of `"DEFERRED"`, `"IMMEDIATE"`, or `"EXCLUSIVE"`, corresponding to the underlying SQLite transaction behaviour, *implicit transaction management* is performed.

If not overridden by the *isolation_level* parameter of `connect()`, the default is `""`, which is an alias for `"DEFERRED"`.

> **Note**
>
> Using `autocommit` to control transaction handling is recommended over using `isolation_level`. `isolation_level` has no effect unless `autocommit` is set to `LEGACY_TRANSACTION_CONTROL` (the default).

row_factory

The initial `row_factory` for `Cursor` objects created from this connection. Assigning to this attribute does not affect the `row_factory` of existing cursors belonging to this connection, only new ones. Is `None` by default, meaning each row is returned as a `tuple`.

See *How to create and use row factories* for more details.

text_factory

A *callable* that accepts a `bytes` parameter and returns a text representation of it. The callable is invoked for SQLite values with the `TEXT` data type. By default, this attribute is set to `str`.

See *How to handle non-UTF-8 text encodings* for more details.

total_changes

Return the total number of database rows that have been modified, inserted, or deleted since the database connection was opened.

Cursor objects

A `Cursor` object represents a database cursor which is used to execute SQL statements, and manage the context of a fetch operation. Cursors are created using `Connection.cursor()`, or by using any of the *connection shortcut methods*.

Cursor objects are *iterators*, meaning that if you `execute()` a `SELECT` query, you can simply iterate over the cursor to fetch the resulting rows :

```
for row in cur.execute("SELECT t FROM data"):
    print(row)
```

class sqlite3.**Cursor**

A `Cursor` instance has the following attributes and methods.

execute(*sql*, *parameters=()*, /)

Execute a single SQL statement, optionally binding Python values using *placeholders*.

Paramètres

- **sql** (`str`) -- A single SQL statement.
- **parameters** (`dict` | *sequence*) -- Python values to bind to placeholders in *sql*. A `dict` if named placeholders are used. A sequence if unnamed placeholders are used. See *How to use placeholders to bind values in SQL queries*.

Lève

ProgrammingError -- If *sql* contains more than one SQL statement.

If `autocommit` is `LEGACY_TRANSACTION_CONTROL`, `isolation_level` is not `None`, *sql* is an `INSERT`, `UPDATE`, `DELETE`, or `REPLACE` statement, and there is no open transaction, a transaction is implicitly opened before executing *sql*.

Deprecated since version 3.12, removed in version 3.14 : `DeprecationWarning` is emitted if *named placeholders* are used and *parameters* is a sequence instead of a `dict`. Starting with Python 3.14, `ProgrammingError` will be raised instead.

Use `executescript()` to execute multiple SQL statements.

executemany(*sql*, *parameters*, /)

For every item in *parameters*, repeatedly execute the *parameterized* DML (Data Manipulation Language) SQL statement *sql*.

Uses the same implicit transaction handling as `execute()`.

Paramètres

— **sql** (`str`) -- A single SQL DML statement.

— **parameters** (*iterable*) -- An iterable of parameters to bind with the placeholders in *sql*. See *How to use placeholders to bind values in SQL queries*.

Lève

ProgrammingError -- If *sql* contains more than one SQL statement, or is not a DML statement.

Exemple :

```
rows = [
    ("row1",),
    ("row2",),
]
# cur is an sqlite3.Cursor object
cur.executemany("INSERT INTO data VALUES(?)", rows)
```

Note

Any resulting rows are discarded, including DML statements with RETURNING clauses.

Deprecated since version 3.12, removed in version 3.14 : `DeprecationWarning` is emitted if *named placeholders* are used and the items in *parameters* are sequences instead of `dict`s. Starting with Python 3.14, `ProgrammingError` will be raised instead.

executescript(*sql_script*, /)

Execute the SQL statements in *sql_script*. If the `autocommit` is `LEGACY_TRANSACTION_CONTROL` and there is a pending transaction, an implicit `COMMIT` statement is executed first. No other implicit transaction control is performed; any transaction control must be added to *sql_script*.

sql_script must be a `string`.

Exemple :

```
# cur is an sqlite3.Cursor object
cur.executescript("""
    BEGIN;
    CREATE TABLE person(firstname, lastname, age);
    CREATE TABLE book(title, author, published);
    CREATE TABLE publisher(name, address);
    COMMIT;
""")
```

fetchone()

If `row_factory` is `None`, return the next row query result set as a `tuple`. Else, pass it to the row factory and return its result. Return `None` if no more data is available.

fetchmany(*size=cursor.arraysize*)

Return the next set of rows of a query result as a `list`. Return an empty list if no more rows are available.

The number of rows to fetch per call is specified by the *size* parameter. If *size* is not given, `arraysize` determines the number of rows to be fetched. If fewer than *size* rows are available, as many rows as are available are returned.

Note there are performance considerations involved with the *size* parameter. For optimal performance, it is usually best to use the arraysize attribute. If the *size* parameter is used, then it is best for it to retain the same value from one `fetchmany()` call to the next.

fetchall()

Return all (remaining) rows of a query result as a `list`. Return an empty list if no rows are available. Note that the `arraysize` attribute can affect the performance of this operation.

close()

Close the cursor now (rather than whenever `__del__` is called).

The cursor will be unusable from this point forward ; a `ProgrammingError` exception will be raised if any operation is attempted with the cursor.

setinputsizes(*sizes*, /)

Required by the DB-API. Does nothing in `sqlite3`.

setoutputsize(*size*, *column=None*, /)

Required by the DB-API. Does nothing in `sqlite3`.

arraysize

Read/write attribute that controls the number of rows returned by `fetchmany()`. The default value is 1 which means a single row would be fetched per call.

connection

Read-only attribute that provides the SQLite database `Connection` belonging to the cursor. A `Cursor` object created by calling `con.cursor()` will have a `connection` attribute that refers to *con* :

```
>>> con = sqlite3.connect(":memory:")
>>> cur = con.cursor()
>>> cur.connection == con
True
>>> con.close()
```

description

Read-only attribute that provides the column names of the last query. To remain compatible with the Python DB API, it returns a 7-tuple for each column where the last six items of each tuple are `None`.

It is set for `SELECT` statements without any matching rows as well.

lastrowid

Read-only attribute that provides the row id of the last inserted row. It is only updated after successful `INSERT` or `REPLACE` statements using the `execute()` method. For other statements, after `executemany()` or `executescript()`, or if the insertion failed, the value of `lastrowid` is left unchanged. The initial value of `lastrowid` is `None`.

> **Note**
>
> Inserts into `WITHOUT ROWID` tables are not recorded.

Modifié dans la version 3.6 : Added support for the `REPLACE` statement.

rowcount

Read-only attribute that provides the number of modified rows for `INSERT`, `UPDATE`, `DELETE`, and `REPLACE` statements ; is `-1` for other statements, including CTE (Common Table Expression) queries. It is only updated by the `execute()` and `executemany()` methods, after the statement has run to completion. This means that any resulting rows must be fetched in order for `rowcount` to be updated.

row_factory

Control how a row fetched from this `Cursor` is represented. If `None`, a row is represented as a `tuple`. Can be set to the included `sqlite3.Row` ; or a *callable* that accepts two arguments, a `Cursor` object and the `tuple` of row values, and returns a custom object representing an SQLite row.

Defaults to what `Connection.row_factory` was set to when the `Cursor` was created. Assigning to this attribute does not affect `Connection.row_factory` of the parent connection.

See *How to create and use row factories* for more details.

Row objects

class sqlite3.**Row**

A `Row` instance serves as a highly optimized `row_factory` for `Connection` objects. It supports iteration, equality testing, `len()`, and *mapping* access by column name and index.

Two `Row` objects compare equal if they have identical column names and values.

See *How to create and use row factories* for more details.

keys()

Return a `list` of column names as `strings`. Immediately after a query, it is the first member of each tuple in `Cursor.description`.

Modifié dans la version 3.5 : Added support of slicing.

Blob objects

class sqlite3.**Blob**

Ajouté dans la version 3.11.

A `Blob` instance is a *file-like object* that can read and write data in an SQLite BLOB. Call `len(blob)` to get the size (number of bytes) of the blob. Use indices and *slices* for direct access to the blob data.

Use the `Blob` as a *context manager* to ensure that the blob handle is closed after use.

```
con = sqlite3.connect(":memory:")
con.execute("CREATE TABLE test(blob_col blob)")
con.execute("INSERT INTO test(blob_col) VALUES(zeroblob(13))")

# Write to our blob, using two write operations:
with con.blobopen("test", "blob_col", 1) as blob:
    blob.write(b"hello, ")
    blob.write(b"world.")
    # Modify the first and last bytes of our blob
    blob[0] = ord("H")
    blob[-1] = ord("!")

# Read the contents of our blob
with con.blobopen("test", "blob_col", 1) as blob:
    greeting = blob.read()

print(greeting)  # outputs "b'Hello, world!'"
con.close()
```

close()

Close the blob.

The blob will be unusable from this point onward. An `Error` (or subclass) exception will be raised if any further operation is attempted with the blob.

read(*length=-1*, /)

Read *length* bytes of data from the blob at the current offset position. If the end of the blob is reached, the data up to EOF (End of File) will be returned. When *length* is not specified, or is negative, `read()` will read until the end of the blob.

write(*data*, /)

Write *data* to the blob at the current offset. This function cannot change the blob length. Writing beyond the end of the blob will raise `ValueError`.

tell()

Return the current access position of the blob.

seek(*offset*, *origin=os.SEEK_SET*, /)

Set the current access position of the blob to *offset*. The *origin* argument defaults to `os.SEEK_SET` (absolute blob positioning). Other values for *origin* are `os.SEEK_CUR` (seek relative to the current position) and `os.SEEK_END` (seek relative to the blob's end).

PrepareProtocol objects

class sqlite3.**PrepareProtocol**

The PrepareProtocol type's single purpose is to act as a **PEP 246** style adaption protocol for objects that can *adapt themselves* to *native SQLite types*.

Exceptions

The exception hierarchy is defined by the DB-API 2.0 (**PEP 249**).

exception sqlite3.**Warning**

This exception is not currently raised by the `sqlite3` module, but may be raised by applications using `sqlite3`, for example if a user-defined function truncates data while inserting. `Warning` is a subclass of `Exception`.

exception sqlite3.**Error**

The base class of the other exceptions in this module. Use this to catch all errors with one single `except` statement. `Error` is a subclass of `Exception`.

If the exception originated from within the SQLite library, the following two attributes are added to the exception :

sqlite_errorcode

The numeric error code from the SQLite API

Ajouté dans la version 3.11.

sqlite_errorname

The symbolic name of the numeric error code from the SQLite API

Ajouté dans la version 3.11.

exception sqlite3.**InterfaceError**

Exception raised for misuse of the low-level SQLite C API. In other words, if this exception is raised, it probably indicates a bug in the `sqlite3` module. `InterfaceError` is a subclass of `Error`.

exception sqlite3.**DatabaseError**

Exception raised for errors that are related to the database. This serves as the base exception for several types of database errors. It is only raised implicitly through the specialised subclasses. `DatabaseError` is a subclass of `Error`.

exception sqlite3.**DataError**

Exception raised for errors caused by problems with the processed data, like numeric values out of range, and strings which are too long. `DataError` is a subclass of `DatabaseError`.

exception sqlite3.**OperationalError**

Exception raised for errors that are related to the database's operation, and not necessarily under the control of the programmer. For example, the database path is not found, or a transaction could not be processed. `OperationalError` is a subclass of `DatabaseError`.

exception sqlite3.**IntegrityError**

Exception raised when the relational integrity of the database is affected, e.g. a foreign key check fails. It is a subclass of `DatabaseError`.

exception sqlite3.**InternalError**

Exception raised when SQLite encounters an internal error. If this is raised, it may indicate that there is a problem with the runtime SQLite library. `InternalError` is a subclass of `DatabaseError`.

exception sqlite3.**ProgrammingError**

Exception raised for sqlite3 API programming errors, for example supplying the wrong number of bindings to a query, or trying to operate on a closed *Connection*. ProgrammingError is a subclass of *DatabaseError*.

exception sqlite3.**NotSupportedError**

Exception raised in case a method or database API is not supported by the underlying SQLite library. For example, setting *deterministic* to True in *create_function()*, if the underlying SQLite library does not support deterministic functions. NotSupportedError is a subclass of *DatabaseError*.

SQLite and Python types

SQLite natively supports the following types : NULL, INTEGER, REAL, TEXT, BLOB.

The following Python types can thus be sent to SQLite without any problem :

Type Python	SQLite type
None	NULL
int	INTEGER
float	REAL
str	TEXT
bytes	BLOB

This is how SQLite types are converted to Python types by default :

SQLite type	Type Python
NULL	None
INTEGER	*int*
REAL	*float*
TEXT	depends on *text_factory*, *str* by default
BLOB	*bytes*

The type system of the sqlite3 module is extensible in two ways : you can store additional Python types in an SQLite database via *object adapters*, and you can let the sqlite3 module convert SQLite types to Python types via *converters*.

Default adapters and converters (deprecated)

Note

The default adapters and converters are deprecated as of Python 3.12. Instead, use the *Adapter and converter recipes* and tailor them to your needs.

The deprecated default adapters and converters consist of :

- An adapter for *datetime.date* objects to *strings* in ISO 8601 format.
- An adapter for *datetime.datetime* objects to strings in ISO 8601 format.
- A converter for *declared* "date" types to *datetime.date* objects.
- A converter for declared "timestamp" types to *datetime.datetime* objects. Fractional parts will be truncated to 6 digits (microsecond precision).

Note

The default "timestamp" converter ignores UTC offsets in the database and always returns a naive *datetime.datetime* object. To preserve UTC offsets in timestamps, either leave converters disabled, or register an offset-aware converter with *register_converter()*.

Obsolète depuis la version 3.12.

Command-line interface

The `sqlite3` module can be invoked as a script, using the interpreter's `-m` switch, in order to provide a simple SQLite shell. The argument signature is as follows :

```
python -m sqlite3 [-h] [-v] [filename] [sql]
```

Type `.quit` or CTRL-D to exit the shell.

-h, --help

Print CLI help.

-v, --version

Print underlying SQLite library version.

Ajouté dans la version 3.12.

12.6.3 How-to guides

How to use placeholders to bind values in SQL queries

SQL operations usually need to use values from Python variables. However, beware of using Python's string operations to assemble queries, as they are vulnerable to SQL injection attacks. For example, an attacker can simply close the single quote and inject `OR TRUE` to select all rows :

```
>>> # Never do this -- insecure!
>>> symbol = input()
' OR TRUE; --
>>> sql = "SELECT * FROM stocks WHERE symbol = '%s'" % symbol
>>> print(sql)
SELECT * FROM stocks WHERE symbol = '' OR TRUE; --'
>>> cur.execute(sql)
```

Instead, use the DB-API's parameter substitution. To insert a variable into a query string, use a placeholder in the string, and substitute the actual values into the query by providing them as a `tuple` of values to the second argument of the cursor's `execute()` method.

An SQL statement may use one of two kinds of placeholders : question marks (qmark style) or named placeholders (named style). For the qmark style, *parameters* must be a *sequence* whose length must match the number of placeholders, or a `ProgrammingError` is raised. For the named style, *parameters* must be an instance of a `dict` (or a subclass), which must contain keys for all named parameters ; any extra items are ignored. Here's an example of both styles :

```
con = sqlite3.connect(":memory:")
cur = con.execute("CREATE TABLE lang(name, first_appeared)")

# This is the named style used with executemany():
data = (
    {"name": "C", "year": 1972},
    {"name": "Fortran", "year": 1957},
    {"name": "Python", "year": 1991},
    {"name": "Go", "year": 2009},
)
cur.executemany("INSERT INTO lang VALUES(:name, :year)", data)

# This is the qmark style used in a SELECT query:
params = (1972,)
cur.execute("SELECT * FROM lang WHERE first_appeared = ?", params)
```

(suite sur la page suivante)

(suite de la page précédente)

```
print(cur.fetchall())
con.close()
```

> **Note**
>
> **PEP 249** numeric placeholders are *not* supported. If used, they will be interpreted as named placeholders.

How to adapt custom Python types to SQLite values

SQLite supports only a limited set of data types natively. To store custom Python types in SQLite databases, *adapt* them to one of the *Python types SQLite natively understands*.

There are two ways to adapt Python objects to SQLite types : letting your object adapt itself, or using an *adapter callable*. The latter will take precedence above the former. For a library that exports a custom type, it may make sense to enable that type to adapt itself. As an application developer, it may make more sense to take direct control by registering custom adapter functions.

How to write adaptable objects

Suppose we have a `Point` class that represents a pair of coordinates, `x` and `y`, in a Cartesian coordinate system. The coordinate pair will be stored as a text string in the database, using a semicolon to separate the coordinates. This can be implemented by adding a `__conform__(self, protocol)` method which returns the adapted value. The object passed to *protocol* will be of type `PrepareProtocol`.

```
class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y

    def __conform__(self, protocol):
        if protocol is sqlite3.PrepareProtocol:
            return f"{self.x};{self.y}"

con = sqlite3.connect(":memory:")
cur = con.cursor()

cur.execute("SELECT ?", (Point(4.0, -3.2),))
print(cur.fetchone()[0])
con.close()
```

How to register adapter callables

The other possibility is to create a function that converts the Python object to an SQLite-compatible type. This function can then be registered using `register_adapter()`.

```
class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y

def adapt_point(point):
    return f"{point.x};{point.y}"

sqlite3.register_adapter(Point, adapt_point)

con = sqlite3.connect(":memory:")
cur = con.cursor()
```

(suite sur la page suivante)

(suite de la page précédente)

```
cur.execute("SELECT ?", (Point(1.0, 2.5),))
print(cur.fetchone()[0])
con.close()
```

How to convert SQLite values to custom Python types

Writing an adapter lets you convert *from* custom Python types *to* SQLite values. To be able to convert *from* SQLite values *to* custom Python types, we use *converters*.

Let's go back to the `Point` class. We stored the x and y coordinates separated via semicolons as strings in SQLite.

First, we'll define a converter function that accepts the string as a parameter and constructs a `Point` object from it.

> **Note**
>
> Converter functions are **always** passed a *bytes* object, no matter the underlying SQLite data type.

```
def convert_point(s):
    x, y = map(float, s.split(b";"))
    return Point(x, y)
```

We now need to tell `sqlite3` when it should convert a given SQLite value. This is done when connecting to a database, using the *detect_types* parameter of *connect()*. There are three options :

- Implicit : set *detect_types* to *PARSE_DECLTYPES*
- Explicit : set *detect_types* to *PARSE_COLNAMES*
- Both : set *detect_types* to `sqlite3.PARSE_DECLTYPES | sqlite3.PARSE_COLNAMES`. Column names take precedence over declared types.

The following example illustrates the implicit and explicit approaches :

```
class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y

    def __repr__(self):
        return f"Point({self.x}, {self.y})"

def adapt_point(point):
    return f"{point.x};{point.y}"

def convert_point(s):
    x, y = list(map(float, s.split(b";")))
    return Point(x, y)

# Register the adapter and converter
sqlite3.register_adapter(Point, adapt_point)
sqlite3.register_converter("point", convert_point)

# 1) Parse using declared types
p = Point(4.0, -3.2)
con = sqlite3.connect(":memory:", detect_types=sqlite3.PARSE_DECLTYPES)
cur = con.execute("CREATE TABLE test(p point)")

cur.execute("INSERT INTO test(p) VALUES(?)", (p,))
cur.execute("SELECT p FROM test")
print("with declared types:", cur.fetchone()[0])
```

(suite sur la page suivante)

(suite de la page précédente)

```
cur.close()
con.close()

# 2) Parse using column names
con = sqlite3.connect(":memory:", detect_types=sqlite3.PARSE_COLNAMES)
cur = con.execute("CREATE TABLE test(p)")

cur.execute("INSERT INTO test(p) VALUES(?)", (p,))
cur.execute('SELECT p AS "p [point]" FROM test')
print("with column names:", cur.fetchone()[0])
cur.close()
con.close()
```

Adapter and converter recipes

This section shows recipes for common adapters and converters.

```
import datetime
import sqlite3

def adapt_date_iso(val):
    """Adapt datetime.date to ISO 8601 date."""
    return val.isoformat()

def adapt_datetime_iso(val):
    """Adapt datetime.datetime to timezone-naive ISO 8601 date."""
    return val.isoformat()

def adapt_datetime_epoch(val):
    """Adapt datetime.datetime to Unix timestamp."""
    return int(val.timestamp())

sqlite3.register_adapter(datetime.date, adapt_date_iso)
sqlite3.register_adapter(datetime.datetime, adapt_datetime_iso)
sqlite3.register_adapter(datetime.datetime, adapt_datetime_epoch)

def convert_date(val):
    """Convert ISO 8601 date to datetime.date object."""
    return datetime.date.fromisoformat(val.decode())

def convert_datetime(val):
    """Convert ISO 8601 datetime to datetime.datetime object."""
    return datetime.datetime.fromisoformat(val.decode())

def convert_timestamp(val):
    """Convert Unix epoch timestamp to datetime.datetime object."""
    return datetime.datetime.fromtimestamp(int(val))

sqlite3.register_converter("date", convert_date)
sqlite3.register_converter("datetime", convert_datetime)
sqlite3.register_converter("timestamp", convert_timestamp)
```

How to use connection shortcut methods

Using the *execute()*, *executemany()*, and *executescript()* methods of the *Connection* class, your code can be written more concisely because you don't have to create the (often superfluous) *Cursor* objects explicitly. Instead, the *Cursor* objects are created implicitly and these shortcut methods return the cursor objects. This way, you can execute a `SELECT` statement and iterate over it directly using only a single call on the *Connection* object.

```
# Create and fill the table.
con = sqlite3.connect(":memory:")
con.execute("CREATE TABLE lang(name, first_appeared)")
data = [
    ("C++", 1985),
    ("Objective-C", 1984),
]
con.executemany("INSERT INTO lang(name, first_appeared) VALUES(?, ?)", data)

# Print the table contents
for row in con.execute("SELECT name, first_appeared FROM lang"):
    print(row)

print("I just deleted", con.execute("DELETE FROM lang").rowcount, "rows")

# close() is not a shortcut method and it's not called automatically;
# the connection object should be closed manually
con.close()
```

How to use the connection context manager

A *Connection* object can be used as a context manager that automatically commits or rolls back open transactions when leaving the body of the context manager. If the body of the `with` statement finishes without exceptions, the transaction is committed. If this commit fails, or if the body of the `with` statement raises an uncaught exception, the transaction is rolled back. If *autocommit* is `False`, a new transaction is implicitly opened after committing or rolling back.

If there is no open transaction upon leaving the body of the `with` statement, or if *autocommit* is `True`, the context manager does nothing.

> **Note**
>
> The context manager neither implicitly opens a new transaction nor closes the connection. If you need a closing context manager, consider using *contextlib.closing()*.

```
con = sqlite3.connect(":memory:")
con.execute("CREATE TABLE lang(id INTEGER PRIMARY KEY, name VARCHAR UNIQUE)")

# Successful, con.commit() is called automatically afterwards
with con:
    con.execute("INSERT INTO lang(name) VALUES(?)", ("Python",))

# con.rollback() is called after the with block finishes with an exception,
# the exception is still raised and must be caught
try:
    with con:
        con.execute("INSERT INTO lang(name) VALUES(?)", ("Python",))
except sqlite3.IntegrityError:
    print("couldn't add Python twice")
```

(suite sur la page suivante)

(suite de la page précédente)

```
# Connection object used as context manager only commits or rollbacks transactions,
# so the connection object should be closed manually
con.close()
```

How to work with SQLite URIs

Some useful URI tricks include :

— Open a database in read-only mode :

```
>>> con = sqlite3.connect("file:tutorial.db?mode=ro", uri=True)
>>> con.execute("CREATE TABLE readonly(data)")
Traceback (most recent call last):
OperationalError: attempt to write a readonly database
>>> con.close()
```

— Do not implicitly create a new database file if it does not already exist; will raise `OperationalError` if unable to create a new file :

```
>>> con = sqlite3.connect("file:nosuchdb.db?mode=rw", uri=True)
Traceback (most recent call last):
OperationalError: unable to open database file
```

— Create a shared named in-memory database :

```
db = "file:mem1?mode=memory&cache=shared"
con1 = sqlite3.connect(db, uri=True)
con2 = sqlite3.connect(db, uri=True)
with con1:
    con1.execute("CREATE TABLE shared(data)")
    con1.execute("INSERT INTO shared VALUES(28)")
res = con2.execute("SELECT data FROM shared")
assert res.fetchone() == (28,)

con1.close()
con2.close()
```

More information about this feature, including a list of parameters, can be found in the SQLite URI documentation.

How to create and use row factories

By default, `sqlite3` represents each row as a `tuple`. If a `tuple` does not suit your needs, you can use the `sqlite3.Row` class or a custom `row_factory`.

While `row_factory` exists as an attribute both on the `Cursor` and the `Connection`, it is recommended to set `Connection.row_factory`, so all cursors created from the connection will use the same row factory.

`Row` provides indexed and case-insensitive named access to columns, with minimal memory overhead and performance impact over a `tuple`. To use `Row` as a row factory, assign it to the `row_factory` attribute :

```
>>> con = sqlite3.connect(":memory:")
>>> con.row_factory = sqlite3.Row
```

Queries now return `Row` objects :

```
>>> res = con.execute("SELECT 'Earth' AS name, 6378 AS radius")
>>> row = res.fetchone()
>>> row.keys()
['name', 'radius']
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> row[0]          # Access by index.
'Earth'
>>> row["name"]     # Access by name.
'Earth'
>>> row["RADIUS"]   # Column names are case-insensitive.
6378
>>> con.close()
```

Note

The `FROM` clause can be omitted in the `SELECT` statement, as in the above example. In such cases, SQLite returns a single row with columns defined by expressions, e.g. literals, with the given aliases `expr AS alias`.

You can create a custom *`row_factory`* that returns each row as a *`dict`*, with column names mapped to values :

```
def dict_factory(cursor, row):
    fields = [column[0] for column in cursor.description]
    return {key: value for key, value in zip(fields, row)}
```

Using it, queries now return a `dict` instead of a `tuple` :

```
>>> con = sqlite3.connect(":memory:")
>>> con.row_factory = dict_factory
>>> for row in con.execute("SELECT 1 AS a, 2 AS b"):
...     print(row)
{'a': 1, 'b': 2}
>>> con.close()
```

The following row factory returns a *named tuple* :

```
from collections import namedtuple

def namedtuple_factory(cursor, row):
    fields = [column[0] for column in cursor.description]
    cls = namedtuple("Row", fields)
    return cls._make(row)
```

`namedtuple_factory()` can be used as follows :

```
>>> con = sqlite3.connect(":memory:")
>>> con.row_factory = namedtuple_factory
>>> cur = con.execute("SELECT 1 AS a, 2 AS b")
>>> row = cur.fetchone()
>>> row
Row(a=1, b=2)
>>> row[0]  # Indexed access.
1
>>> row.b   # Attribute access.
2
>>> con.close()
```

With some adjustments, the above recipe can be adapted to use a *`dataclass`*, or any other custom class, instead of a *`namedtuple`*.

How to handle non-UTF-8 text encodings

By default, `sqlite3` uses `str` to adapt SQLite values with the `TEXT` data type. This works well for UTF-8 encoded text, but it might fail for other encodings and invalid UTF-8. You can use a custom `text_factory` to handle such cases.

Because of SQLite's flexible typing, it is not uncommon to encounter table columns with the `TEXT` data type containing non-UTF-8 encodings, or even arbitrary data. To demonstrate, let's assume we have a database with ISO-8859-2 (Latin-2) encoded text, for example a table of Czech-English dictionary entries. Assuming we now have a `Connection` instance `con` connected to this database, we can decode the Latin-2 encoded text using this `text_factory` :

```
con.text_factory = lambda data: str(data, encoding="latin2")
```

For invalid UTF-8 or arbitrary data in stored in `TEXT` table columns, you can use the following technique, borrowed from the unicode-howto :

```
con.text_factory = lambda data: str(data, errors="surrogateescape")
```

Note

The `sqlite3` module API does not support strings containing surrogates.

Voir aussi

unicode-howto

12.6.4 Explanation

Transaction control

`sqlite3` offers multiple methods of controlling whether, when and how database transactions are opened and closed. *Transaction control via the autocommit attribute* is recommended, while *Transaction control via the isolation_level attribute* retains the pre-Python 3.12 behaviour.

Transaction control via the `autocommit` attribute

The recommended way of controlling transaction behaviour is through the `Connection.autocommit` attribute, which should preferably be set using the *autocommit* parameter of `connect()`.

It is suggested to set *autocommit* to `False`, which implies **PEP 249**-compliant transaction control. This means :

- `sqlite3` ensures that a transaction is always open, so `connect()`, `Connection.commit()`, and `Connection.rollback()` will implicitly open a new transaction (immediately after closing the pending one, for the latter two). `sqlite3` uses `BEGIN DEFERRED` statements when opening transactions.
- Transactions should be committed explicitly using `commit()`.
- Transactions should be rolled back explicitly using `rollback()`.
- An implicit rollback is performed if the database is `close()`-ed with pending changes.

Set *autocommit* to `True` to enable SQLite's autocommit mode. In this mode, `Connection.commit()` and `Connection.rollback()` have no effect. Note that SQLite's autocommit mode is distinct from the **PEP 249**-compliant `Connection.autocommit` attribute ; use `Connection.in_transaction` to query the low-level SQLite autocommit mode.

Set *autocommit* to `LEGACY_TRANSACTION_CONTROL` to leave transaction control behaviour to the `Connection.isolation_level` attribute. See *Transaction control via the isolation_level attribute* for more information.

Transaction control via the `isolation_level` attribute

> **Note**
>
> The recommended way of controlling transactions is via the `autocommit` attribute. See *Transaction control via the autocommit attribute*.

If `Connection.autocommit` is set to `LEGACY_TRANSACTION_CONTROL` (the default), transaction behaviour is controlled using the `Connection.isolation_level` attribute. Otherwise, `isolation_level` has no effect.

If the connection attribute `isolation_level` is not `None`, new transactions are implicitly opened before `execute()` and `executemany()` executes `INSERT`, `UPDATE`, `DELETE`, or `REPLACE` statements ; for other statements, no implicit transaction handling is performed. Use the `commit()` and `rollback()` methods to respectively commit and roll back pending transactions. You can choose the underlying SQLite transaction behaviour — that is, whether and what type of `BEGIN` statements `sqlite3` implicitly executes – via the `isolation_level` attribute.

If `isolation_level` is set to `None`, no transactions are implicitly opened at all. This leaves the underlying SQLite library in autocommit mode, but also allows the user to perform their own transaction handling using explicit SQL statements. The underlying SQLite library autocommit mode can be queried using the `in_transaction` attribute.

The `executescript()` method implicitly commits any pending transaction before execution of the given SQL script, regardless of the value of `isolation_level`.

Modifié dans la version 3.6 : `sqlite3` used to implicitly commit an open transaction before DDL statements. This is no longer the case.

Modifié dans la version 3.12 : The recommended way of controlling transactions is now via the `autocommit` attribute.

CHAPITRE 13

Compression de donnée et archivage

Les modules documentés dans ce chapitre implémentent les algorithmes de compression *zlib*, *gzip*, *bzip2* et *lzma*, ainsi que la création d'archives *ZIP* et *tar*. Voir aussi *Archiving operations* fourni par le module `shutil`.

13.1 `zlib` --- Compression compatible with `gzip`

Pour des applications nécessitant de compresser des données, les fonctions de ce module permettent la compression et la décompression via la bibliothèque *zlib*. La bibliothèque *zlib* a sa propre page web sur https://www.zlib.net. Il existe des incompatibilités connues entre le module Python et les versions de la bibliothèque *zlib* plus anciennes que la 1.1.3 ; 1.1.3 contient une faille de sécurité et nous recommandons d'utiliser plutôt la version 1.1.4 ou plus récente.

Les fonctions *zlib* recèlent de nombreuses options et il est nécessaire de suivre un ordre précis. Cette documentation n'a pas pour but de couvrir la globalité des possibilités. Aussi, veuillez consulter le manuel *zlib* en ligne sur http://www.zlib.net/manual.html pour compléter davantage son utilisation.

Pour lire ou écrire des fichiers `.gz` veuillez consulter le module `gzip`.

Les exceptions et fonctions disponibles dans ce module sont :

exception `zlib.`**`error`**

Exception levée lors d'erreurs de compression et de décompression.

`zlib.`**`adler32`**(*data*[, *value*])

Calcule une somme de contrôle Adler-32 de *data* (une somme de contrôle Adler-32 est aussi fiable qu'un CRC32 mais peut être calculée bien plus rapidement). Le résultat produit est un entier non signé de 32-bit. Si *value* est défini, il devient la valeur initiale de la somme de contrôle ; sinon une valeur par défaut de 1 est utilisée. Définir *value* permet de calculer une somme de contrôle continue pendant la concaténation de plusieurs entrées. Cet algorithme n'a aucune garantie cryptographique puissante, et ne doit pas être utilisé ni pour l'authentification, ni pour des signatures numériques. Conçu comme un algorithme de somme de contrôle, il n'est pas adapté pour une utilisation sous forme de clé de hachage générique.

Modifié dans la version 3.0 : Renvoie une valeur non-signée.

`zlib.`**`compress`**(*data*, */*, *level=-1*, *wbits=MAX_WBITS*)

Compresses the bytes in *data*, returning a bytes object containing compressed data. *level* is an integer from `0` to `9` or `-1` controlling the level of compression ; `1` (Z_BEST_SPEED) is fastest and produces the least compression, `9` (Z_BEST_COMPRESSION) is slowest and produces the most. `0` (Z_NO_COMPRESSION) is no

compression. The default value is `-1` (Z_DEFAULT_COMPRESSION). Z_DEFAULT_COMPRESSION represents a default compromise between speed and compression (currently equivalent to level 6).

L'argument *wbits* contrôle la taille du tampon d'historique ("*window size*") utilisé lors de la compression, et si un en-tête et un bloc final seront inclus. Il accepte plusieurs intervalles de valeurs, et vaut `15` (MAX_WBITS) par défaut :

— De +9 à +15 : le logarithme binaire de la taille du tampon, par conséquent compris entre 512 et 32 768. Des valeurs plus grandes produisent de meilleures compressions aux dépens d'une utilisation mémoire plus grande. Le résultat final inclus des en-têtes et des blocs spécifiques à *zlib*.

— De `-9` à `-15` : utilise la valeur absolue de *wbits* comme logarithme binaire de la taille du tampon, et ne produit pas d'entêtes ni de bloc final.

— De +25 à +31 = 16 + (9 à 15) : utilise les 4 bits de poids faible comme logarithme binaire de la taille du tampon, tout en incluant un entête **gzip** et une somme de contrôle finale.

Raises the `error` exception if any error occurs.

Modifié dans la version 3.6 : *level* peut maintenant être passé par son nom.

Modifié dans la version 3.11 : The *wbits* parameter is now available to set window bits and compression type.

`zlib.`**`compressobj`**(*level=-1*, *method=DEFLATED*, *wbits=MAX_WBITS*, *memLevel=DEF_MEM_LEVEL*, *strategy=Z_DEFAULT_STRATEGY*[, *zdict*])

Renvoie un objet "compresseur", à utiliser pour compresser des flux de données qui ne peuvent pas être stockés entièrement en mémoire.

level est le taux de compression -- un entier compris entre `0` et `9` ou `-1`. La valeur `1` est la plus rapide avec taux de compression minimal, tandis que la valeur `9` est la plus lente mais produit une compression maximale. `0` ne produit aucune compression. La valeur par défaut est `-1` (*Z_DEFAULT_COMPRESSION*). La constante *Z_DEFAULT_COMPRESSION* équivaut à un bon compromis par défaut entre rapidité et bonne compression (valeur équivalente au niveau 6).

method définit l'algorithme de compression. Pour le moment, la seule valeur acceptée est `DEFLATED`.

The *wbits* parameter controls the size of the history buffer (or the "window size"), and what header and trailer format will be used. It has the same meaning as *described for compress()*.

L'argument *memLevel* permet d'ajuster la quantité de mémoire utilisée pour stocker l'état interne de la compression. Les valeurs valides sont comprises entre `1` et `9`. Des valeurs plus élevées occupent davantage de mémoire, mais sont plus rapides et produisent des sorties plus compressées.

strategy permet d'ajuster l'algorithme de compression. Les valeurs possibles sont `Z_DEFAULT_STRATEGY`, `Z_FILTERED`, et `Z_HUFFMAN_ONLY`, `Z_RLE` (*zlib* 1.2.0.1) et `Z_FIXED` (*zlib* 1.2.2.2).

zdict est un dictionnaire de compression prédéfini. C'est une séquence d'octets (tel qu'un objet `bytes`) contenant des sous-séquences attendues régulièrement dans les données à compresser. Les sous-séquences les plus fréquentes sont à placer à la fin du dictionnaire.

Modifié dans la version 3.3 : Ajout du paramètre *zdict*.

`zlib.`**`crc32`**(*data*[, *value*])

Calcule la somme de contrôle CRC (*Cyclic Redundancy Check* en anglais) de l'argument *data*. Il renvoie un entier non signé de 32 bits. Si l'argument *value* est présent, il permet de définir la valeur de départ de la somme de contrôle. Sinon, la valeur par défaut est 0. L'argument *value* permet de calculer la somme de contrôle glissante d'une concaténation de données. L'algorithme n'est pas fort d'un point de vue cryptographique, et ne doit pas être utilisé pour l'authentification ou des signatures numériques. Cet algorithme a été conçu pour être exploité comme un algorithme de somme de contrôle, ce n'est pas un algorithme de hachage générique.

Modifié dans la version 3.0 : Renvoie une valeur non-signée.

`zlib.`**`decompress`**(*data*, */*, *wbits=MAX_WBITS*, *bufsize=DEF_BUF_SIZE*)

Décompresse les octets de *data*, renvoyant un objet *bytes* contenant les données décompressées. Le paramètre *wbits* dépend du format des données compressées, et est abordé plus loin. Si l'argument *bufsize* est défini, il est utilisé comme taille initiale du tampon de sortie. En cas d'erreur, l'exception `error` est levée.

L'argument *wbits* contrôle la taille du tampon d'historique ("*window size*") utilisé lors de la compression, et si un en-tête et un bloc final sont attendus. Similaire au paramètre de `compressobj()`, mais accepte une gamme

plus large de valeurs :

— De +8 à +15 : logarithme binaire pour la taille du tampon. L'entrée doit contenir un en-tête et un bloc *zlib*.
— 0 : détermine automatiquement la taille du tampon à partir de l'en-tête *zlib*. Géré uniquement depuis *zlib* 1.2.3.5.
— De `-8` à `-15` : utilise la valeur absolue de *wbits* comme logarithme binaire pour la taille du tampon. L'entrée doit être un flux brut, sans en-tête ni bloc final.
— De +24 à +31 = 16 + (8 à 15) : utilise les 4 de poids faible comme logarithme binaire pour la taille du tampon. L'entrée doit contenir un en-tête *gzip* et son bloc final.
— De +40 à +47 = 32 + (8 à 15) : utilise les 4 bits de poids faible comme logarithme binaire pour la taille du tampon, et accepte automatiquement les formats *zlib* ou *gzip*.

Lors de la décompression d'un flux, la taille du tampon ne doit pas être inférieure à la taille initialement utilisée pour compresser le flux. L'utilisation d'une valeur trop petite peut déclencher une exception `error`. La valeur par défaut *wbits* correspond à une taille élevée du tampon et nécessite d'y adjoindre un en-tête *zlib* et son bloc final.

L'argument *bufsize* correspond à la taille initiale du tampon utilisé pour contenir les données décompressées. Si plus d'espace est nécessaire, la taille du tampon sera augmentée au besoin, donc vous n'avez pas besoin de deviner la valeur exacte. Un réglage précis n'économisera que quelques appels à `malloc()`.

Modifié dans la version 3.6 : *wbits* et *bufsize* peuvent être utilisés comme arguments nommés.

`zlib.`**`decompressobj`**(*wbits=MAX_WBITS*[, *zdict*])

Renvoie un objet "décompresseur", à utiliser pour décompresser des flux de données qui ne rentrent pas entièrement en mémoire.

Le paramètre *wbits* contrôle la taille du tampon, et détermine quel format d'en-tête et de bloc sont prévus. Il a la même signification que décrit pour *decompress()*.

Le paramètre *zdict* définit un dictionnaire de compression prédéfini. S'il est fourni, il doit être identique au dictionnaire utilisé par le compresseur, à l'origine des données à décompresser.

> **Note**
>
> Si *zdict* est un objet modifiable (tel qu'un `bytearray`, par exemple), vous ne devez pas modifier son contenu entre l'appel à la fonction `decompressobj()` et le premier appel à la méthode `decompress()` du décompresseur.

Modifié dans la version 3.3 : Ajout du paramètre *zdict*.

Les objets de compression gèrent les méthodes suivantes :

`Compress.`**`compress`**(*data*)

Compresse *data* et renvoie au moins une partie des données compressées sous forme d'objet *bytes*. Ces données doivent être concaténées à la suite des appels précédant à `compress()`. Certaines entrées peuvent être conservées dans des tampons internes pour un traitement ultérieur.

`Compress.`**`flush`**([*mode*])

Toutes les entrées mises en attente sont traitées et un objet *bytes* contenant la sortie des données compressées restantes est renvoyé. L'argument *mode* accepte l'une des constantes suivantes : `Z_NO_FLUSH`, `Z_PARTIAL_FLUSH`, `Z_SYNC_FLUSH`, `Z_FULL_FLUSH`, `Z_BLOCK` (*zlib* 1.2.3.4), et `Z_FINISH`, par défaut `Z_FINISH`. Sauf `Z_FINISH`, toutes les constantes permettent de compresser d'autres chaînes d'octets, tandis que `Z_FINISH` finalise le flux compressé et bloque toute autre tentative de compression. Suite à l'appel de la méthode `flush()` avec l'argument *mode* défini à `Z_FINISH`, la méthode `compress()` ne peut plus être rappelée. Il ne reste plus qu'à supprimer l'objet.

`Compress.`**`copy`**()

Renvoie une copie de l'objet "compresseur". Utile pour compresser efficacement un ensemble de données qui partagent un préfixe initial commun.

Modifié dans la version 3.8 : Les objets compresseurs gère maintenant `copy.copy()` et `copy.deepcopy()`.

Les objets décompresseurs prennent en charge les méthodes et attributs suivants :

`Decompress.unused_data`

Un objet *bytes* contenant tous les octets restants après les données compressées. Il vaut donc `b""` tant que des données compressées sont disponibles. Si toute la chaîne d'octets ne contient que des données compressées, il vaut toujours `b""`, un objet *bytes* vide.

`Decompress.unconsumed_tail`

Un objet *bytes* contenant toutes les données non-traitées par le dernier appel à la méthode `decompress()`, à cause d'un dépassement de la limite du tampon de données décompressées. Ces données n'ont pas encore été traitées par la bibliothèque *zlib*, vous devez donc les envoyer (potentiellement en y concaténant encore des données) par un appel à la méthode `decompress()` pour obtenir une sortie correcte.

`Decompress.eof`

Booléen qui signale si la fin du flux compressé est atteint.

This makes it possible to distinguish between a properly formed compressed stream, and an incomplete or truncated one.

Ajouté dans la version 3.3.

`Decompress.decompress(data, max_length=0)`

Décompresse *data*, renvoie un objet *bytes*, contenant au moins une partie des données décompressées. Ce résultat doit être concaténé aux résultats des appels précédents à `decompress()`. Des données d'entrée peuvent être conservées dans les tampons internes pour un traitement ultérieur.

Si le paramètre optionnel *max_length* est différent de zéro alors la valeur renvoyée n'est pas plus grande que *max_length*. Cela peut amener à une décompression partielle. Les données non-encore décompressées sont stockées dans l'attribut `unconsumed_tail`. Cette chaîne d'octets doit être transmise à un appel ultérieur à `decompress()`. Si *max_length* vaut zéro, la totalité de l'entrée est décompressée, et l'attribut `unconsumed_tail` reste vide.

Modifié dans la version 3.6 : *max_length* peut être utilisé comme un argument nommé.

`Decompress.flush([length])`

Toutes les entrées en attente sont traitées, et un objet *bytes* est renvoyé, contenant le reste des données à décompresser. Après l'appel à `flush()`, la méthode `decompress()` ne peut pas être rappelée. Il ne reste qu'à détruire l'objet.

Le paramètre optionnel *length* définit la taille initiale du tampon de sortie.

`Decompress.copy()`

Renvoie une copie du décompresseur. Vous pouvez l'utiliser pour sauvegarder l'état de la décompression en cours, afin de pouvoir revenir rapidement à cet endroit plus tard.

Modifié dans la version 3.8 : Ajout de `copy.copy()` et de `copy.deepcopy()` au support de décompression d'objets.

Des informations relatives à la version de la bibliothèque *zlib* utilisée sont disponibles via les constantes suivantes :

`zlib.ZLIB_VERSION`

Version de la bibliothèque *zlib* utilisée lors de la compilation du module. Elle peut être différente de la bibliothèque *zlib* actuellement utilisée par le système, qui est consultable par `ZLIB_RUNTIME_VERSION`.

`zlib.ZLIB_RUNTIME_VERSION`

Chaîne contenant la version de la bibliothèque *zlib* actuellement utilisée par l'interpréteur.

Ajouté dans la version 3.3.

Voir aussi

Module `gzip`
Lire et écrire des fichiers au format `gzip`.

http://www.zlib.net
Page officielle de la bibliothèque *zlib*.

http://www.zlib.net/manual.html
La documentation de *zlib* explique le sens et l'utilisation des nombreuses fonctions fournies par la bibliothèque.

13.2 `gzip` --- Support for `gzip` files

Code source : Lib/gzip.py

Ce module fournit une interface simple pour compresser et décompresser des fichiers tout comme le font les programmes GNU **gzip** et **gunzip**.

La compression de données est fournie par le module `zlib`.

Le module `gzip` fournit la classe `GzipFile` ainsi que les fonctions pratiques `open()`, `compress()` et `decompress()`. La classe `GzipFile` lit et écrit des fichiers au format **gzip**, compressant et décompressant automatiquement les données pour qu'elles aient l'apparence d'un objet *file object* ordinaire.

Notez que les formats de fichier supplémentaires qui peuvent être décompressés par les programmes **gzip** et **gunzip**, comme ceux produits par le programmes **compress** et **pack**, ne sont pas gérés par ce module.

Le module définit les éléments suivants :

`gzip.`**`open`**(*filename*, *mode='rb'*, *compresslevel=9*, *encoding=None*, *errors=None*, *newline=None*)

Ouvre un fichier compressé en *gzip* en mode binaire ou texte, renvoie un objet *file object*.

L'argument *filename* peut être un nom de fichier (un objet `str` ou `bytes`) ou un objet fichier existant que l'on peut lire, ou où l'on peut écrire.

L'argument *mode* peut-être `'r'`, `'rb'`, `'a'`, `'ab'`, `'w'`, `'wb'`, `'x'` ou `'xb'` pour le mode binaire ou `'rt'`, `'at'`, `'wt'`, ou `'xt'` pour le mode texte. Le mode par défaut est `'rb'`.

L'argument *compresslevel* est un entier de 0 à 9, comme pour le constructeur de la classe `GzipFile`.

En mode binaire, cette fonction est équivalente au constructeur de la classe `GzipFile` : `GzipFile(filename, mode, compresslevel)`. Dans ce cas, les arguments *encoding*, *errors* et *newline* ne doivent pas être fournis.

En mode texte, un objet `GzipFile` est créé et empaqueté dans une instance de `io.TextIOWrapper` avec l'encodage, la gestion d'erreur et les fins de ligne spécifiés.

Modifié dans la version 3.3 : Ajout de la prise en charge de *filename* en tant qu'objet *file*, du mode texte et des arguments *encoding*, *errors* et *newline*.

Modifié dans la version 3.4 : Ajout de la prise en charge des modes `'x'`, `'xb'` et `'xt'`.

Modifié dans la version 3.6 : Accepte un *path-like object*.

exception `gzip.`**`BadGzipFile`**

An exception raised for invalid gzip files. It inherits from `OSError`. `EOFError` and `zlib.error` can also be raised for invalid gzip files.

Ajouté dans la version 3.8.

class `gzip.`**`GzipFile`**(*filename=None*, *mode=None*, *compresslevel=9*, *fileobj=None*, *mtime=None*)

Constructor for the `GzipFile` class, which simulates most of the methods of a *file object*, with the exception of the `truncate()` method. At least one of *fileobj* and *filename* must be given a non-trivial value.

La nouvelle instance de classe est basée sur *fileobj* qui peut être un fichier usuel, un objet `io.BytesIO` ou tout autre objet qui simule un fichier. *fileobj* est par défaut à *None*, dans ce cas *filename* est ouvert afin de fournir un objet fichier.

Quand *fileobj* n'est pas à `None`, l'argument *filename* est uniquement utilisé pour être inclus dans l'entête du fichier **gzip**, qui peut inclure le nom original du fichier décompressé. Il est par défaut défini avec le nom de

fichier de *fileobj* s'il est discernable, sinon il est par défaut défini à une chaîne de caractères vide et dans ce cas le nom du fichier orignal n'est pas inclus dans l'entête.

The *mode* argument can be any of `'r'`, `'rb'`, `'a'`, `'ab'`, `'w'`, `'wb'`, `'x'`, or `'xb'`, depending on whether the file will be read or written. The default is the mode of *fileobj* if discernible ; otherwise, the default is `'rb'`. In future Python releases the mode of *fileobj* will not be used. It is better to always specify *mode* for writing.

Notez que le fichier est toujours ouvert en mode binaire. Pour ouvrir un fichier compressé en mode texte, utilisez la fonction `open()` (ou empaquetez votre classe `GzipFile` avec un `io.TextIOWrapper`).

L'argument *compresslevel* est un entier de `0` à `9` contrôlant le niveau de compression, `1` est le plus rapide et produit la compression la plus faible et `9` est le plus rapide et produit la compression la plus élevée. `0` désactive la compression. Par défaut à `9`.

The optional *mtime* argument is the timestamp requested by gzip. The time is in Unix format, i.e., seconds since 00 :00 :00 UTC, January 1, 1970. If *mtime* is omitted or `None`, the current time is used. Use *mtime* = 0 to generate a compressed stream that does not depend on creation time.

See below for the `mtime` attribute that is set when decompressing.

Calling a `GzipFile` object's `close()` method does not close *fileobj*, since you might wish to append more material after the compressed data. This also allows you to pass an `io.BytesIO` object opened for writing as *fileobj*, and retrieve the resulting memory buffer using the `io.BytesIO` object's `getvalue()` method.

`GzipFile` supports the `io.BufferedIOBase` interface, including iteration and the `with` statement. Only the `truncate()` method isn't implemented.

La classe `GzipFile` fournit aussi la méthode et l'attribut suivant :

peek(*n*)

Lit *n* octets non compressés sans avancer la position dans le fichier. Au plus une seule lecture sur le flux compressé est faite pour satisfaire l'appel. Le nombre d'octets retournés peut être supérieur ou inférieur au nombre demandé.

> **Note**
>
> Bien que l'appel à `peek()` ne change pas la position dans le fichier de l'objet `GzipFile`, il peut changer la position de l'objet de fichier sous-jacent (par exemple, si l'instance de `GzipFile` a été construite avec le paramètre *fileobj*).

Ajouté dans la version 3.2.

mode

`'rb'` for reading and `'wb'` for writing.

Modifié dans la version 3.13 : In previous versions it was an integer `1` or `2`.

mtime

When decompressing, this attribute is set to the last timestamp in the most recently read header. It is an integer, holding the number of seconds since the Unix epoch (00 :00 :00 UTC, January 1, 1970). The initial value before reading any headers is `None`.

name

The path to the gzip file on disk, as a `str` or `bytes`. Equivalent to the output of `os.fspath()` on the original input path, with no other normalization, resolution or expansion.

Modifié dans la version 3.1 : Ajout de la prise en charge du mot-clef `with`, ainsi que de l'argument *mtime* du constructeur et de l'attribut `mtime`.

Modifié dans la version 3.2 : Ajout de la prise en charge des fichiers non navigables ou commençant par des octets nuls.

Modifié dans la version 3.3 : La méthode `io.BufferedIOBase.read1()` est désormais implémentée.

Modifié dans la version 3.4 : Ajout de la prise en charge des modes `'x'` et `'xb'`.

Modifié dans la version 3.5 : Ajout de la prise en charge de l'écriture d'objets *bytes-like objects* arbitraires. La méthode `read()` accepte désormais un argument de valeur `None`.

Modifié dans la version 3.6 : Accepte un *path-like object*.

Obsolète depuis la version 3.9 : Opening `GzipFile` for writing without specifying the *mode* argument is deprecated.

Modifié dans la version 3.12 : Remove the `filename` attribute, use the `name` attribute instead.

`gzip.`**`compress`**(*data*, *compresslevel=9*, **, mtime=0*)

Compress the *data*, returning a `bytes` object containing the compressed data. *compresslevel* and *mtime* have the same meaning as in the `GzipFile` constructor above, but *mtime* defaults to 0 for reproducible output.

Ajouté dans la version 3.2.

Modifié dans la version 3.8 : Added the *mtime* parameter for reproducible output.

Modifié dans la version 3.11 : Speed is improved by compressing all data at once instead of in a streamed fashion. Calls with *mtime* set to `0` are delegated to `zlib.compress()` for better speed. In this situation the output may contain a gzip header "OS" byte value other than 255 "unknown" as supplied by the underlying zlib implementation.

Modifié dans la version 3.13 : The gzip header OS byte is guaranteed to be set to 255 when this function is used as was the case in 3.10 and earlier.

Modifié dans la version 3.14 : The *mtime* parameter now defaults to 0 for reproducible output. For the previous behaviour of using the current time, pass `None` to *mtime*.

`gzip.`**`decompress`**(*data*)

Decompress the *data*, returning a `bytes` object containing the uncompressed data. This function is capable of decompressing multi-member gzip data (multiple gzip blocks concatenated together). When the data is certain to contain only one member the `zlib.decompress()` function with *wbits* set to 31 is faster.

Ajouté dans la version 3.2.

Modifié dans la version 3.11 : Speed is improved by decompressing members at once in memory instead of in a streamed fashion.

13.2.1 Exemples d'utilisation

Exemple montrant comment lire un fichier compressé :

```
import gzip
with gzip.open('/home/joe/file.txt.gz', 'rb') as f:
    file_content = f.read()
```

Exemple montrant comment créer un fichier GZIP :

```
import gzip
content = b"Lots of content here"
with gzip.open('/home/joe/file.txt.gz', 'wb') as f:
    f.write(content)
```

Exemple montrant comment compresser dans un GZIP un fichier existant :

```
import gzip
import shutil
with open('/home/joe/file.txt', 'rb') as f_in:
    with gzip.open('/home/joe/file.txt.gz', 'wb') as f_out:
        shutil.copyfileobj(f_in, f_out)
```

Exemple montrant comment compresser dans un GZIP un binaire dans une chaîne :

```
import gzip
s_in = b"Lots of content here"
s_out = gzip.compress(s_in)
```

Voir aussi

Module *zlib*
Le module de compression de données de base nécessaire pour gérer le format de fichier **gzip**.

13.2.2 Interface en ligne de commande

The *gzip* module provides a simple command line interface to compress or decompress files.

Once executed the *gzip* module keeps the input file(s).

Modifié dans la version 3.8 : Add a new command line interface with a usage. By default, when you will execute the CLI, the default compression level is 6.

Options de la ligne de commande

file
If *file* is not specified, read from *sys.stdin*.

--fast
Indicates the fastest compression method (less compression).

--best
Indicates the slowest compression method (best compression).

-d, --decompress
Decompress the given file.

-h, --help
Affiche le message d'aide.

13.3 bz2 --- Support for bzip2 compression

Code Source : Lib/bz2.py

Ce module fournit une interface complète pour compresser et décompresser les données en utilisant l'algorithme de compression *bzip2*.

Le module *bz2* contient :
- La fonction *open()* et la classe *BZ2File* pour lire et écrire des fichiers compressés.
- Les classes *BZ2Compressor* et *BZ2Decompressor* pour la (dé)compression incrémentielle.
- Les fonctions *compress()* et *decompress()* pour la (dé)compression en une seule fois.

13.3.1 (Dé)compression de fichiers

bz2.**open**(*filename*, *mode='rb'*, *compresslevel=9*, *encoding=None*, *errors=None*, *newline=None*)
Ouvre un fichier compressé par *bzip2* en mode binaire ou texte, le renvoyant en *file object*.

Tout comme avec le constructeur pour la classe *BZ2File*, l'argument *filename* peut être un nom de fichier réel (un objet *str* ou *bytes*), ou un objet fichier existant à lire ou à écrire.

L'argument *mode* peut valoir `'r'`, `'rb'`, `'w'`, `'wb'`, `'x'`, `'xb'`, `'a'` ou `'ab'` pour le mode binaire, ou `'rt'`, `'wt'`, `'xt'` ou `'at'` pour le mode texte. Il vaut par défaut `'rb'`.

L'argument *compresslevel* est un entier de 1 à 9, comme pour le constructeur `BZ2File`.

Pour le mode binaire, cette fonction est équivalente au constructeur `BZ2File`:`BZ2File(filename, mode, compresslevel=compresslevel)`. Dans ce cas, les arguments *encoding*, *errors* et *newline* arguments ne doivent pas être fournis.

Pour le mode texte, un objet `BZ2File` est créé et encapsulé dans une instance `io.TextIOWrapper` avec l'encodage spécifié, le comportement de gestion des erreurs et les fins de ligne.

Ajouté dans la version 3.3.

Modifié dans la version 3.4 : Le mode `'x'` (création exclusive) est ajouté.

Modifié dans la version 3.6 : Accepte un *path-like object*.

class `bz2.`**`BZ2File`**(*filename*, *mode='r'*, *, *compresslevel=9*)

Ouvre un fichier *bzip2* en mode binaire.

Si *filename* est un objet `str` ou `bytes`, ouvre le nom de fichier directement. Autrement, *filename* doit être un *file object*, qui est utilisé pour lire ou écrire les données compressées.

L'argument *mode* peut être soit `'r'` pour lire (par défaut), `'w'` pour écraser, `'x'` pour créer exclusivement, ou `'a'` pour ajouter. Ils peuvent également être écrits respectivement comme `'rb'`, `'wb'`, `'xb'` et `'ab'`.

Si *filename* est un objet fichier (plutôt que le nom de fichier réel), le mode `'w'` ne tronque pas le fichier, mais équivaut à `'a'`.

Si *mode* est `'w'` ou `'a'`, *compresslevel* peut être un entier entre `1` et `9` spécifiant le niveau de compression : `1` utilise la compression la moins forte, et `9` (par défaut) la compression la plus forte.

Si *mode* est `'r'`, le fichier d'entrée peut être la concaténation de plusieurs flux compressés.

`BZ2File` provides all of the members specified by the `io.BufferedIOBase`, except for `detach()` and `truncate()`. Iteration and the `with` statement are supported.

`BZ2File` also provides the following methods and attributes :

peek([*n*])

Renvoie des données en mémoire tampon sans avancer la position du fichier. Au moins un octet de donnée (sauf l'EOF) est renvoyé. Le nombre exact d'octets renvoyés n'est pas spécifié.

> **Note**
>
> Bien que l'appel à la méthode `peek()` ne change pas la position du fichier de la classe `BZ2File`, il peut changer la position de l'objet fichier sous-jacent (e.g. si la classe `BZ2File` a été construite en passant un objet fichier à *filename*).

Ajouté dans la version 3.3.

fileno()

Return the file descriptor for the underlying file.

Ajouté dans la version 3.3.

readable()

Return whether the file was opened for reading.

Ajouté dans la version 3.3.

seekable()

Return whether the file supports seeking.

Ajouté dans la version 3.3.

writable()

Return whether the file was opened for writing.

Ajouté dans la version 3.3.

read1(*size=-1*)

Read up to *size* uncompressed bytes, while trying to avoid making multiple reads from the underlying stream. Reads up to a buffer's worth of data if size is negative.

Returns `b''` if the file is at EOF.

Ajouté dans la version 3.3.

readinto(*b*)

Read bytes into *b*.

Returns the number of bytes read (0 for EOF).

Ajouté dans la version 3.3.

mode

`'rb'` for reading and `'wb'` for writing.

Ajouté dans la version 3.13.

name

The bzip2 file name. Equivalent to the `name` attribute of the underlying *file object*.

Ajouté dans la version 3.13.

Modifié dans la version 3.1 : La prise en charge de l'instruction `with` a été ajoutée.

Modifié dans la version 3.3 : La gestion de *filename* comme *file object* au lieu d'un nom de fichier réel a été ajoutée.

Le mode `'a'` (ajout) a été ajouté, avec la prise en charge de la lecture des fichiers *multiflux*.

Modifié dans la version 3.4 : Le mode `'x'` (création exclusive) est ajouté.

Modifié dans la version 3.5 : La méthode `read()` accepte maintenant un argument `None`.

Modifié dans la version 3.6 : Accepte un *path-like object*.

Modifié dans la version 3.9 : Le paramètre *buffering* a été supprimé. Il était ignoré et obsolète depuis Python 3.0. Passez un objet fichier déjà ouvert si vous voulez contrôler la façon dont le fichier est ouvert.

Le paramètre *compresslevel* est devenu un paramètre exclusivement nommé.

Modifié dans la version 3.10 : Cette classe n'est plus protégée d'un accès concurrent malheureux par plusieurs lecteurs ou compresseurs. C'était déjà le cas depuis toujours des classes analogues dans les modules `gzip` et `lzma`.

13.3.2 (Dé)compression incrémentielle

class bz2.**BZ2Compressor**(*compresslevel=9*)

Crée un nouvel objet compresseur. Cet objet peut être utilisé pour compresser les données de manière incrémentielle. Pour une compression en une seule fois, utilisez à la place la fonction `compress()`.

compresslevel, s'il est fourni, doit être un entier entre `1` et `9`. Sa valeur par défaut est `9`.

compress(*data*)

Fournit la donnée à l'objet compresseur. Renvoie un bloc de données compressées si possible, ou autrement une chaîne d'octet vide.

Quand vous avez fini de fournir des données au compresseur, appelez la méthode `flush()` pour finir le processus de compression.

flush()

Finit le processus de compression. Renvoie la donnée compressée restante dans les tampons internes.

L'objet compresseur ne peut pas être utilisé après que cette méthode a été appelée.

class bz2.**BZ2Decompressor**

Crée un nouvel objet décompresseur. Cet objet peut être utilisé pour décompresser les données de manière incrémentielle. Pour une compression en une seule fois, utilisez à la place la fonction `decompress()`.

> **Note**
>
> Cette classe ne gère pas de manière transparente les entrées contenant plusieurs flux compressés, à la différence de `decompress()` et `BZ2File`. Si vous avez besoin de décompresser une entrée *multiflux* avec la classe `BZ2Decompressor`, vous devez utiliser un nouveau décompresseur pour chaque flux.

decompress(*data*, *max_length=-1*)

Décompresse *data* (un *bytes-like object*), renvoyant une donnée non compressée en tant que chaîne d'octets. Certaines de ces *data* peuvent être mises en interne en tampon, pour un usage lors d'appels ultérieurs par la méthode `decompress()`. La donnée renvoyée doit être concaténée avec la sortie des appels précédents à la méthode `decompress()`.

Si *max_length* est positif, renvoie au plus *max_length* octets de données compressées. Si la limite est atteinte et que d'autres sorties peuvent être produites, l'attribut `needs_input` est positionné sur `False`. Dans ce cas, lors de l'appel suivant à la méthode `decompress()`, vous pouvez fournir `b''` dans *data* afin d'obtenir la suite de la sortie.

Si toutes les données entrées ont été décompressées et renvoyées (soit parce qu'il y avait moins de *max_length* octets, ou parce que *max_length* était négatif), l'attribut `needs_input` sera configuré sur `True`.

Attempting to decompress data after the end of stream is reached raises an `EOFError`. Any data found after the end of the stream is ignored and saved in the `unused_data` attribute.

Modifié dans la version 3.5 : Ajout du paramètre *max_length*.

eof

`True` si le marqueur de fin de flux a été atteint.

Ajouté dans la version 3.3.

unused_data

Donnée trouvée après la fin du flux compressé.

Si l'attribut est accédé avant que la fin du flux ait été atteint, sa valeur sera `b''`.

needs_input

`False` si la méthode `decompress()` peut fournir plus de données décompressées avant l'acquisition d'une nouvelle entrée non compressée.

Ajouté dans la version 3.5.

13.3.3 (Dé)compression en une fois

bz2.**compress**(*data*, *compresslevel=9*)

Compresse *data*, un *bytes-like object*.

compresslevel, s'il est fourni, doit être un entier entre `1` et `9`. Sa valeur par défaut est `9`.

Pour la compression incrémentielle, utilisez à la place la classe `BZ2Compressor`.

bz2.**decompress**(*data*)

Décompresse *data*, un *bytes-like object*.

Si *data* est la concaténation de multiples flux compressés, décompresse tous les flux.

Pour une décompression incrémentielle, utilisez à la place la classe `BZ2Decompressor`.

Modifié dans la version 3.3 : Prise en charge des entrées *multiflux*.

13.3.4 Exemples d'utilisation

Ci-dessous, nous présentons quelques exemples typiques de l'utilisation du module `bz2`.

Utilise les fonctions `compress()` et `decompress()` pour démontrer une compression aller-retour :

```
>>> import bz2
>>> data = b"""\
... Donec rhoncus quis sapien sit amet molestie. Fusce scelerisque vel augue
... nec ullamcorper. Nam rutrum pretium placerat. Aliquam vel tristique lorem,
... sit amet cursus ante. In interdum laoreet mi, sit amet ultrices purus
... pulvinar a. Nam gravida euismod magna, non varius justo tincidunt feugiat.
... Aliquam pharetra lacus non risus vehicula rutrum. Maecenas aliquam leo
... felis. Pellentesque semper nunc sit amet nibh ullamcorper, ac elementum
... dolor luctus. Curabitur lacinia mi ornare consectetur vestibulum."""
>>> c = bz2.compress(data)
>>> len(data) / len(c)  # Data compression ratio
1.513595166163142
>>> d = bz2.decompress(c)
>>> data == d  # Check equality to original object after round-trip
True
```

Utilise la classe `BZ2Compressor` pour une compression incrémentielle :

```
>>> import bz2
>>> def gen_data(chunks=10, chunksize=1000):
...     """Yield incremental blocks of chunksize bytes."""
...     for _ in range(chunks):
...         yield b"z" * chunksize
...
>>> comp = bz2.BZ2Compressor()
>>> out = b""
>>> for chunk in gen_data():
...     # Provide data to the compressor object
...     out = out + comp.compress(chunk)
...
>>> # Finish the compression process.  Call this once you have
>>> # finished providing data to the compressor.
>>> out = out + comp.flush()
```

The example above uses a very "nonrandom" stream of data (a stream of `b"z"` chunks). Random data tends to compress poorly, while ordered, repetitive data usually yields a high compression ratio.

Écriture et lecture en mode binaire d'un fichier compressé avec *bzip2* :

```
>>> import bz2
>>> data = b"""\
... Donec rhoncus quis sapien sit amet molestie. Fusce scelerisque vel augue
... nec ullamcorper. Nam rutrum pretium placerat. Aliquam vel tristique lorem,
... sit amet cursus ante. In interdum laoreet mi, sit amet ultrices purus
... pulvinar a. Nam gravida euismod magna, non varius justo tincidunt feugiat.
... Aliquam pharetra lacus non risus vehicula rutrum. Maecenas aliquam leo
... felis. Pellentesque semper nunc sit amet nibh ullamcorper, ac elementum
... dolor luctus. Curabitur lacinia mi ornare consectetur vestibulum."""
>>> with bz2.open("myfile.bz2", "wb") as f:
...     # Write compressed data to file
...     unused = f.write(data)
...
>>> with bz2.open("myfile.bz2", "rb") as f:
```

(suite sur la page suivante)

(suite de la page précédente)

```
...     # Decompress data from file
...     content = f.read()
...
>>> content == data  # Check equality to original object after round-trip
True
```

13.4 lzma --- Compression using the LZMA algorithm

Ajouté dans la version 3.3.

Code source : Lib/lzma.py

Ce module fournit des classes et des fonctions utiles pour compresser et décompresser des données en utilisant l'algorithme de compression LZMA. Ce module inclut aussi aussi une interface prenant en charge les fichiers `.xz` et son format originel `.lzma` utilisés par l'utilitaire **xz**, ainsi que les flux bruts compressés.

The interface provided by this module is very similar to that of the `bz2` module. Note that `LZMAFile` and `bz2.BZ2File` are *not* thread-safe, so if you need to use a single `LZMAFile` instance from multiple threads, it is necessary to protect it with a lock.

exception lzma.**LZMAError**

Cette exception est levée dès lors qu'une erreur survient pendant la compression ou la décompression, ou pendant l'initialisation de l'état de la compression/décompression.

13.4.1 Lire et écrire des fichiers compressés

lzma.**open**(*filename*, *mode='rb'*, ***, *format=None*, *check=-1*, *preset=None*, *filters=None*, *encoding=None*, *errors=None*, *newline=None*)

Open an LZMA-compressed file in binary or text mode, returning a *file object*.

L'argument *nom de fichier* peut être soit le nom d'un fichier à créer (donné pour `str`, `bytes` ou un objet *path-like*), dont le fichier nommé reste ouvert, ou soit un objet fichier existant à lire ou à écrire.

L'argument *mode* peut être n'importe quel argument suivant : `"r"`, `"rb"`, `"w"`, `"wb"`, `"x"`, `"xb"`, `"a"` ou `"ab"` pour le mode binaire, ou `"rt"`, `"wt"`, `"xt"`, ou `"at"` pour le mode texte. La valeur par défaut est `"rb"`.

Quand un fichier est ouvert pour le lire, les arguments *format* et *filters* ont les mêmes significations que pour la `LZMADecompressor`. Par conséquent, les arguments *check* et *preset* ne devront pas être sollicités.

Dès ouverture d'un fichier pour l'écriture, les arguments *format*, *check*, *preset* et *filters* ont le même sens que dans la `LZMACompressor`.

Pour le mode binaire, cette fonction équivaut au constructeur de la `LZMAFile` : `LZMAFile(filename, mode, ...)`. Dans ce cas précis, les arguments *encoding*, *errors* et *newline* ne sont pas accessibles.

For text mode, a `LZMAFile` object is created, and wrapped in an `io.TextIOWrapper` instance with the specified encoding, error handling behavior, and line ending(s).

Modifié dans la version 3.4 : Support ajouté pour les modes `"x"`, `"xb"` et `"xt"`.

Modifié dans la version 3.6 : Accepte un *path-like object*.

class lzma.**LZMAFile**(*filename=None*, *mode='r'*, ***, *format=None*, *check=-1*, *preset=None*, *filters=None*)

Ouvre un fichier LZMA compressé en mode binaire.

An `LZMAFile` can wrap an already-open *file object*, or operate directly on a named file. The *filename* argument specifies either the file object to wrap, or the name of the file to open (as a `str`, `bytes` or *path-like* object). When wrapping an existing file object, the wrapped file will not be closed when the `LZMAFile` is closed.

L'argument *mode* peut être soit `"r"` pour la lecture (défaut), `"w"` pour la ré-écriture, `"x"` pour la création exclusive, ou `"a"` pour l'insertion. Elles peuvent aussi être écrites de la façon suivante : `"rb"`, `"wb"`, `"xb"` et `"ab"` respectivement.

If *filename* is a file object (rather than an actual file name), a mode of `"w"` does not truncate the file, and is instead equivalent to `"a"`.

Dès l'ouverture d'un fichier pour être lu, le fichier d'entrée peut être le résultat d'une concaténation de plusieurs flux distincts et compressés. Ceux-ci sont décodés de manière transparente en un seul flux logique.

Quand un fichier est ouvert pour le lire, les arguments *format* et *filters* ont les mêmes significations que pour la `LZMADecompressor`. Par conséquent, les arguments *check* et *preset* ne devront pas être sollicités.

Dès ouverture d'un fichier pour l'écriture, les arguments *format*, *check*, *preset* et *filters* ont le même sens que dans la `LZMACompressor`.

`LZMAFile` supports all the members specified by `io.BufferedIOBase`, except for `detach()` and `truncate()`. Iteration and the `with` statement are supported.

The following method and attributes are also provided :

peek(*size=-1*)

Renvoie la donnée en mémoire-tampon sans progression de la position du fichier. Au moins un octet de donnée sera renvoyé, jusqu'à ce que l'EOF soit atteinte. Le nombre exact d'octets renvoyés demeure indéterminé (l'argument *taille* est ignoré).

> **Note**
>
> While calling `peek()` does not change the file position of the `LZMAFile`, it may change the position of the underlying file object (e.g. if the `LZMAFile` was constructed by passing a file object for *filename*).

mode

`'rb'` for reading and `'wb'` for writing.

Ajouté dans la version 3.13.

name

The lzma file name. Equivalent to the `name` attribute of the underlying *file object*.

Ajouté dans la version 3.13.

Modifié dans la version 3.4 : Added support for the `"x"` and `"xb"` modes.

Modifié dans la version 3.5 : La méthode `read()` accepte maintenant un argument `None`.

Modifié dans la version 3.6 : Accepte un *path-like object*.

13.4.2 Compresser et décompresser une donnée en mémoire

class lzma.**LZMACompressor**(*format=FORMAT_XZ*, *check=-1*, *preset=None*, *filters=None*)

Créé un objet compresseur, qui peut être utilisé pour compresser incrémentalement une donnée.

Pour une façon plus adaptée de compresser un seul extrait de donnée, voir `compress()`.

L'argument *format* définit quel format de conteneur sera mis en œuvre. Les valeurs possibles sont :

— **`FORMAT_XZ` : Le format du conteneur `.xz`.**
  C'est le format par défaut.

— **`FORMAT_ALONE` : L'ancien format du conteneur `.lzma`.**
  Ce format est davantage limité que `.xz` --il ne supporte pas les vérifications d'intégrité ou les filtres multiples.

— **`FORMAT_RAW` : Un flux de données brut, n'utilisant aucun format de conteneur.**
  Ce format spécifique ne prend pas en charge les vérifications d'intégrité et exige systématiquement la définition d'une chaîne de filtrage personnalisée (à la fois pour la compression et la décompression).

Par ailleurs, les données compressées par ce biais ne peuvent pas être décompressées par l'usage de `FORMAT_AUTO` (voir : `LZMADecompressor`).

L'argument *check* détermine le type de vérification d'intégrité à exploiter avec la donnée compressée. Cette vérification est déclenchée lors de la décompression, pour garantir que la donnée n'a pas été corrompue. Les valeurs possibles sont :

- `CHECK_NONE` : Pas de vérification d'intégrité. C'est la valeur par défaut (et la seule valeur acceptable) pour `FORMAT_ALONE` et `FORMAT_RAW`.
- `CHECK_CRC32` : Vérification par Redondance Cyclique 32-bit (*Cyclic Redundancy Check*).
- `CHECK_CRC64` : Vérification par Redondance Cyclique 64-bit (*Cyclic Redundancy Check*). Valeur par défaut pour `FORMAT_XZ`.
- `CHECK_SHA256` : Algorithme de Hachage Sécurisé 256-bit (*Secure Hash Algorithm*).

Si le type de vérification n'est pas supporté par le système, une erreur de type `LZMAError` est levée.

Les réglages de compression peuvent être définis soit comme un pré-réglage de niveau de compression (avec l'argument *preset*) ; soit de façon détaillée comme une chaîne particulière de filtres (avec l'argument *filters*).

L'argument *preset* (s'il est fourni) doit être un entier compris entre `0` et `9` (inclus), éventuellement relié à OR avec la constante `PRESET_EXTREME`. Si aucun *preset* ni *filters* ne ont définis, le comportement par défaut consiste à utiliser la `PRESET_DEFAULT` (niveau par défaut : `6`). Des pré-réglages plus élevés entraîne une sortie plus petite, mais rend le processus de compression plus lent.

> **Note**
>
> En plus d'être plus gourmande en CPU, la compression avec des préréglages plus élevés nécessite beaucoup plus de mémoire (et produit des résultats qui nécessitent plus de mémoire pour décompresser). Par exemple, avec le préréglage `9`, l'objet d'une `LZMACompressor` peut dépasser largement les 800 Mo. Pour cette raison, il est généralement préférable de respecter le préréglage par défaut.

L'argument *filters* (s'il est défini) doit être un critère de la chaîne de filtrage. Voir *Préciser des chaînes de filtre personnalisées* pour plus de précisions.

compress(*data*)

Une *data* compressée (un objet `bytes`), renvoie un objet `bytes` contenant une donnée compressée pour au moins une partie de l'entrée. Certaine *data* peuvent être mise en tampon, pour être utiliser lors de prochains appels par `compress()` et `flush()`. La donnée renvoyée pourra être concaténée avec la sortie d'appels précédents vers la méthode `compress()`.

flush()

Conclut l'opération de compression, en renvoyant l'objet `bytes` constitué de toutes les données stockées dans les tampons interne du compresseur.

The compressor cannot be used after this method has been called.

class lzma.**LZMADecompressor**(*format=FORMAT_AUTO*, *memlimit=None*, *filters=None*)

Créé un objet de décompression, pour décompresser de façon incrémentale une donnée.

Pour un moyen plus pratique de décompresser un flux compressé complet en une seule fois, voir `decompress()`.

L'argument *format* spécifie le format du conteneur à utiliser. La valeur par défaut est `FORMAT_AUTO` pouvant à la fois décompresser les fichiers `.xz` et `.lzma`. D'autres valeurs sont possibles comme `FORMAT_XZ`, `FORMAT_ALONE`, et `FORMAT_RAW`.

L'argument *memlimit* spécifie une limite (en octets) sur la quantité de mémoire que le décompresseur peut utiliser. Lorsque cet argument est utilisé, la décompression échouera avec une `LZMAError` s'il n'est pas possible de décompresser l'entrée dans la limite mémoire disponible.

L'argument *filters* spécifie la chaîne de filtrage utilisée pour créer le flux décompressé. Cet argument est requis si *format* est `FORMAT_RAW`, mais ne doit pas être utilisé pour d'autres formats. Voir *Préciser des chaînes de filtre personnalisées* pour plus d'informations sur les chaînes de filtrage.

> **Note**
>
> Cette classe ne gère pas de manière transparente les entrées contenant plusieurs flux compressés, contrairement à `decompress()` et `LZMAFile`. Pour décompresser une entrée multi-flux avec `LZMADecompressor`, vous devez créer un nouveau décompresseur à chaque flux.

decompress(*data*, *max_length=-1*)

Décompresse *data* (un *bytes-like object*), renvoyant une donnée non compressée en tant que chaîne d'octets. Certaines de ces *data* peuvent être mises en interne en tampon, pour un usage lors d'appels ultérieurs par la méthode `decompress()`. La donnée renvoyée doit être concaténée avec la sortie des appels précédents à la méthode `decompress()`.

Si *max_length* est positif, renvoie au plus *max_length* octets de données compressées. Si la limite est atteinte et que d'autres sorties peuvent être produites, l'attribut `needs_input` est positionné sur `False`. Dans ce cas, lors de l'appel suivant à la méthode `decompress()`, vous pouvez fournir `b''` dans *data* afin d'obtenir la suite de la sortie.

Si toutes les données entrées ont été décompressées et renvoyées (soit parce qu'il y avait moins de *max_length* octets, ou parce que *max_length* était négatif), l'attribut `needs_input` sera configuré sur `True`.

Attempting to decompress data after the end of stream is reached raises an `EOFError`. Any data found after the end of the stream is ignored and saved in the `unused_data` attribute.

Modifié dans la version 3.5 : Ajout du paramètre *max_length*.

check

L'ID de la vérification d'intégrité exploité par le flux entrant. Il s'agit de `CHECK_UNKNOWN` tant que ce flux a été décodé pour déterminer quel type de vérification d'intégrité à été utilisé.

eof

`True` si le marqueur de fin de flux a été atteint.

unused_data

Donnée trouvée après la fin du flux compressé.

Avant d'atteindre la fin du flux, ce sera `b""`.

needs_input

`False` si la méthode `decompress()` peut fournir plus de données décompressées avant l'acquisition d'une nouvelle entrée non compressée.

Ajouté dans la version 3.5.

`lzma`.**compress**(*data*, *format=FORMAT_XZ*, *check=-1*, *preset=None*, *filters=None*)

data compressée (un objet `bytes`), renvoyant une donnée compressée comme un objet `bytes`.

Voir `LZMACompressor` ci-dessus pour une description des arguments *format*, *check*, *preset* et *filters*.

`lzma`.**decompress**(*data*, *format=FORMAT_AUTO*, *memlimit=None*, *filters=None*)

Décompresse *data* (un objet `bytes`), et retourne la donnée décompressée sous la forme d'un objet `bytes`.

Si *data* est le résultat de la concaténation de plusieurs flux compressés et distincts , il les décompresse tous, et retourne les résultats concaténés.

Voir `LZMADecompressor` ci-dessus pour une description des arguments *format*, *memlimit* et *filters*.

13.4.3 Divers

`lzma`.**is_check_supported**(*check*)

Return `True` if the given integrity check is supported on this system.

`CHECK_NONE` et `CHECK_CRC32` sont toujours pris en charge. `CHECK_CRC64` et `CHECK_SHA256` peuvent être indisponibles si vous utilisez une version de **liblzma** compilée avec des possibilités restreintes.

13.4.4 Préciser des chaînes de filtre personnalisées

Une chaîne de filtres est une séquence de dictionnaires, où chaque dictionnaire contient l'ID et les options pour chaque filtre. Le moindre dictionnaire contient la clé `"id"` et peut aussi contenir d'autres clés pour préciser chaque options relative au filtre déclaré. Les ID valides des filtres sont définies comme suit :

- Filtres de compression :
  - `FILTER_LZMA1` (à utiliser avec `FORMAT_ALONE`)
  - `FILTER_LZMA2` (à utiliser avec `FORMAT_XZ` et `FORMAT_RAW`)
- Filtre Delta :
  - `FILTER_DELTA`
- Filtres Branch-Call-Jump (BCJ) :
  - `FILTER_X86`
  - `FILTER_IA64`
  - `FILTER_ARM`
  - `FILTER_ARMTHUMB`
  - `FILTER_POWERPC`
  - `FILTER_SPARC`

Une chaîne de filtres peut contenir jusqu'à 4 filtres, et ne peut pas être vide. Le dernier filtre de cette chaîne devra être un filtre de compression, et tous les autres doivent être des filtres delta ou BCJ.

Les filtres de compression contiennent les options suivantes (définies comme entrées additionnelles dans le dictionnaire qui représente le filtre) :

- `preset` : Un pré-réglage à exploiter comme une source de valeurs par défaut pour les options qui ne sont pas explicitement définies.
- `dict_size` : La taille du dictionnaire en octets. Comprise entre 4 Ko et 1.5 Go (inclus).
- `lc` : Nombre de bits dans le contexte littéral.
- `lp` : Nombre de bits dans la position littérale. La somme `lc + lp` devra être au moins 4.
- `pb` : Nombre de bits à cette position ; au moins 4.
- `mode` : `MODE_FAST` ou `MODE_NORMAL`.
- `nice_len` : Ce qui devra être pris en compte comme "longueur appréciable" pour une recherche. Il devra être 273 ou moins.
- `mf` : Quel type d'index de recherche à utiliser -- `MF_HC3`, `MF_HC4`, `MF_BT2`, `MF_BT3`, ou `MF_BT4`.
- `depth` : Profondeur maximum de la recherche, utilisée par l'index de recherche. 0 (défaut) signifie une sélection automatique basée sur des options de filtres différents.

Le filtre delta stocke les différences entre octets, induisant davantage d'entrées répétitives pour le compresseur, selon les circonstances. Il support une option, `dist`. Ce paramètre définit la distance entre les octets à soustraire. Par défaut : 1, soit la différence entre des octets adjacents.

Les filtres BCJ sont conçus pour être appliqués sur du langage machine. Ils convertissent les branches relatives, les appels et les sauts dans le code à des fins d'adressage strict, dans le but d'augmenter la redondance mise en jeu par le compresseur. Ils ne supportent qu'une seule option : `start_offset`, pour définir l'adresse où sera déclenché le début de la donnée d'entrée. Par défaut : 0.

13.4.5 Exemples

Lire un fichier compressé :

```
import lzma
with lzma.open("file.xz") as f:
    file_content = f.read()
```

Créer un fichier compressé :

```
import lzma
data = b"Insert Data Here"
with lzma.open("file.xz", "w") as f:
    f.write(data)
```

Compresser des données en mémoire :

```
import lzma
data_in = b"Insert Data Here"
data_out = lzma.compress(data_in)
```

Compression incrémentale :

```
import lzma
lzc = lzma.LZMACompressor()
out1 = lzc.compress(b"Some data\n")
out2 = lzc.compress(b"Another piece of data\n")
out3 = lzc.compress(b"Even more data\n")
out4 = lzc.flush()
# Concatenate all the partial results:
result = b"".join([out1, out2, out3, out4])
```

Écrire des données compressées dans un fichier préalablement ouvert :

```
import lzma
with open("file.xz", "wb") as f:
    f.write(b"This data will not be compressed\n")
    with lzma.open(f, "w") as lzf:
        lzf.write(b"This *will* be compressed\n")
    f.write(b"Not compressed\n")
```

Créer un fichier compressé en utilisant une chaîne de filtre personnalisée :

```
import lzma
my_filters = [
    {"id": lzma.FILTER_DELTA, "dist": 5},
    {"id": lzma.FILTER_LZMA2, "preset": 7 | lzma.PRESET_EXTREME},
]
with lzma.open("file.xz", "w", filters=my_filters) as f:
    f.write(b"blah blah blah")
```

13.5 `zipfile` --- Work with ZIP archives

Source code : Lib/zipfile/

Le format de fichier ZIP est une archive et un standard de compression couramment utilisés. Ce module fournit des outils pour créer, écrire, ajouter des données à et lister un fichier ZIP. L'utilisation avancée de ce module requiert une certaine compréhension du format, comme défini dans PKZIP Application Note.

Ce module ne gère pas pour l'instant les fichiers ZIP multi-disque. Il gère les fichiers ZIP qui utilisent les extensions ZIP64 (c'est-à-dire des fichiers d'une taille supérieure à 4 Go). Il gère le chiffrement d'archives ZIP chiffrées, mais il ne peut pas pour l'instant créer de fichier chiffré. Le déchiffrement est extrêmement lent car il est implémenté uniquement en Python plutôt qu'en C.

Le module définit les éléments suivants :

exception `zipfile.`**`BadZipFile`**

Erreur levée en cas de fichier ZIP non valide.

Ajouté dans la version 3.2.

exception `zipfile.`**`BadZipfile`**

Alias de `BadZipFile`, pour la compatibilité avec les versions de Python précédentes.

Obsolète depuis la version 3.2.

exception zipfile.**LargeZipFile**

Erreur levée quand un fichier ZIP nécessite la fonctionnalité ZIP64 mais qu'elle n'a pas été activée.

class zipfile.**ZipFile**

Classe pour lire et écrire des fichiers ZIP. Voir la section *Objets ZipFile* pour les détails du constructeur.

class zipfile.**Path**

Class that implements a subset of the interface provided by `pathlib.Path`, including the full `importlib.resources.abc.Traversable` interface.

Ajouté dans la version 3.8.

class zipfile.**PyZipFile**

Classe pour créer des archives ZIP contenant des bibliothèques Python.

class zipfile.**ZipInfo**(*filename='NoName'*, *date_time=(1980, 1, 1, 0, 0, 0)*)

Classe utilisée pour représenter les informations d'un membre d'une archive. Les instances de cette classe sont retournées par les méthodes `getinfo()` et `infolist()` des objets `ZipFile`. La plupart des utilisateurs du module `zipfile` n'ont pas besoin de créer ces instances mais d'utiliser celles créées par ce module. *filename* doit être le nom complet du membre de l'archive et *date_time* doit être un sextuplet décrivant la date de dernière modification du fichier ; les champs sont décrits dans la section *Objets ZipInfo*.

Modifié dans la version 3.13 : A public `compress_level` attribute has been added to expose the formerly protected `_compresslevel`. The older protected name continues to work as a property for backwards compatibility.

_for_archive(*archive*)

Resolve the date_time, compression attributes, and external attributes to suitable defaults as used by `ZipFile.writestr()`.

Returns self for chaining.

Ajouté dans la version 3.14.

zipfile.**is_zipfile**(*filename*)

Retourne `True` si *filename* est un fichier ZIP valide basé sur son nombre magique, sinon retourne `False`. *filename* peut aussi être un fichier ou un objet fichier-compatible.

Modifié dans la version 3.1 : Gestion des objets fichier et fichier-compatibles.

zipfile.**ZIP_STORED**

Constante numérique pour un membre d'une archive décompressée.

zipfile.**ZIP_DEFLATED**

Constante numérique pour la méthode habituelle de compression de ZIP. Nécessite le module `zlib`.

zipfile.**ZIP_BZIP2**

Constante numérique pour la méthode de compressions BZIP2. Nécessite le module `bz2`.

Ajouté dans la version 3.3.

zipfile.**ZIP_LZMA**

Constante numérique pour la méthode de compressions LZMA. Nécessite le module `lzma`.

Ajouté dans la version 3.3.

> **Note**
>
> La spécification du format de fichier ZIP inclut la gestion de la compression BZIP2 depuis 2001 et LZMA depuis 2006. Néanmoins, certains outils (comme certaines versions de Python) ne gèrent pas ces méthodes de compression et peuvent soit totalement refuser de traiter le fichier ZIP soit ne pas extraire certains fichiers.

Voir aussi

PKZIP Application Note
Documentation sur le format de fichier ZIP par Phil Katz, créateur du format et des algorithmes utilisés.

Info-ZIP Home Page
Informations sur les programmes et les bibliothèques de développement d'archivage ZIP du projet Info-ZIP.

13.5.1 Objets ZipFile

class zipfile.**ZipFile**(*file*, *mode='r'*, *compression=ZIP_STORED*, *allowZip64=True*, *compresslevel=None*, *, *strict_timestamps=True*, *metadata_encoding=None*)

Ouvre un fichier ZIP, où *file* peut être un chemin vers un fichier (une chaîne de caractères), un objet fichier-compatible ou un objet chemin-compatible *path-like object*.

Le paramètre *mode* doit être `r` pour lire un fichier existant, `w` pour tronquer et écrire un nouveau fichier, `a` pour ajouter des données à la fin d'un fichier existant ou `x` pour créer et écrire exclusivement un nouveau fichier. Si *mode* est à `x` et *file* fait référence à un fichier existant, une exception `FileExistsError` est levée. Si *mode* est à `a` et *file* fait référence à un fichier ZIP existant, alors des fichiers supplémentaires y seront ajoutés. Si *file* ne fait pas référence à un fichier ZIP, alors une nouvelle archive ZIP est ajoutée au fichier, afin de prévoir le cas d'ajouter une archive ZIP à un autre fichier (comme par exemple `python.exe`). Si *mode* est à `r` ou `a`, le fichier doit être navigable.

Le paramètre *compression* est la méthode de compression ZIP à utiliser lors de l'écriture de l'archive et doit être défini à `ZIP_STORED`, `ZIP_DEFLATED`, `ZIP_BZIP2` ou `ZIP_LZMA`; les valeurs non reconnues lèveront une exception `NotImplementedError`. Si `ZIP_DEFLATED`, `ZIP_BZIP2` ou `ZIP_LZMA` est spécifié mais le module correspondant (`zlib`, `bz2` ou `lzma`) n'est pas disponible, une exception `RuntimeError` est levée. Est défini par défaut à `ZIP_STORED`.

Si *allowZip64* est à `True` (par défaut), *zipfile* crée des fichiers ZIP utilisant les extensions ZIP64 quand le fichier ZIP est plus grand que 4 Go. S'il est à `False`, `zipfile` lève une exception quand le fichier ZIP nécessiterait les extensions ZIP64.

Le paramètre *compresslevel* contrôle le niveau de compression à utiliser lors de l'écriture des fichiers dans l'archive. Avec `ZIP_STORED` ou `ZIP_LZMA`, cela est sans effet. Avec `ZIP_DEFLATED` les entiers de `0` à `9` sont acceptés (voir `zlib` pour plus d'informations). Avec `ZIP_BZIP2` les entiers de `1` à `9` sont acceptés (voir `bz2` pour plus d'informations).

Les fichiers ZIP plus anciens que le 1er janvier 1980 sont autorisés lorsque l'argument *strict_timestamps* vaut `False`, moyennant de les voir datés du 1er janvier 1980. De même pour les fichiers datés d'après le 31 décembre 2107 qui voient leur horodatage fixé au 31 décembre 2107.

When mode is `'r'`, *metadata_encoding* may be set to the name of a codec, which will be used to decode metadata such as the names of members and ZIP comments.

Si le fichier est créé avec le mode `'w'`, `'x'` ou `'a'` et ensuite `fermé` sans ajouter de fichiers à l'archive, la structure appropriée pour un fichier archive ZIP vide sera écrite dans le fichier.

ZipFile est aussi un gestionnaire de contexte et gère ainsi la déclaration `with`. Dans l'exemple, *myzip* est fermé à la fin de la déclaration `with` --- même si une exception est levée :

```
with ZipFile('spam.zip', 'w') as myzip:
    myzip.write('eggs.txt')
```

Note

metadata_encoding is an instance-wide setting for the ZipFile. It is not currently possible to set this on a per-member basis.

> This attribute is a workaround for legacy implementations which produce archives with names in the current locale encoding or code page (mostly on Windows). According to the .ZIP standard, the encoding of metadata may be specified to be either IBM code page (default) or UTF-8 by a flag in the archive header. That flag takes precedence over *metadata_encoding*, which is a Python-specific extension.

Modifié dans la version 3.2 : Ajout de la possibilité d'utiliser `ZipFile` comme un gestionnaire de contexte.

Modifié dans la version 3.3 : Ajout de la gestion de la compression `bzip2` et `lzma`.

Modifié dans la version 3.4 : Les extensions ZIP64 sont activées par défaut.

Modifié dans la version 3.5 : Ajout de la gestion de l'écriture dans des flux non navigables. Ajout de la gestion du mode `x`.

Modifié dans la version 3.6 : Auparavant, une simple exception `RuntimeError` était levée pour des valeurs de compression non reconnues.

Modifié dans la version 3.6.2 : Le paramètre *file* accepte un objet fichier-compatible *path-like object*.

Modifié dans la version 3.7 : Ajout du paramètre *compresslevel*.

Modifié dans la version 3.8 : The *strict_timestamps* keyword-only parameter.

Modifié dans la version 3.11 : Added support for specifying member name encoding for reading metadata in the zipfile's directory and file headers.

`ZipFile.`**`close`**`()`

Ferme l'archive. Vous devez appeler `close()` avant de terminer votre programme ou des informations essentielles n'y seront pas enregistrées.

`ZipFile.`**`getinfo`**`(`*name*`)`

Retourne un objet `ZipInfo` avec les informations du membre *name* de l'archive. Appeler `getinfo()` pour un nom non contenu dans l'archive lève une exception `KeyError`.

`ZipFile.`**`infolist`**`()`

Retourne une liste contenant un objet `ZipInfo` pour chaque membre de l'archive. Les objets ont le même ordre que leurs entrées dans le fichier ZIP présent sur disque s'il s'agissait d'une archive préexistante.

`ZipFile.`**`namelist`**`()`

Retourne une liste des membres de l'archive indexés par leur nom.

`ZipFile.`**`open`**`(`*name*, *mode='r'*, *pwd=None*, **, force_zip64=False*`)`

Access a member of the archive as a binary file-like object. *name* can be either the name of a file within the archive or a `ZipInfo` object. The *mode* parameter, if included, must be `'r'` (the default) or `'w'`. *pwd* is the password used to decrypt encrypted ZIP files as a `bytes` object.

`open()` est aussi un gestionnaire de contexte et gère ainsi la déclaration `with` :

```
with ZipFile('spam.zip') as myzip:
    with myzip.open('eggs.txt') as myfile:
        print(myfile.read())
```

With *mode* `'r'` the file-like object (`ZipExtFile`) is read-only and provides the following methods : `read()`, `readline()`, `readlines()`, `seek()`, `tell()`, `__iter__()`, `__next__()`. These objects can operate independently of the ZipFile.

Avec `mode='w'` un descripteur de fichier en écriture est retourné, gérant la méthode `write()`. Quand le descripteur d'un fichier inscriptible est ouvert, tenter de lire ou écrire d'autres fichiers dans le fichier ZIP lève une exception `ValueError`.

In both cases the file-like object has also attributes `name`, which is equivalent to the name of a file within the archive, and `mode`, which is `'rb'` or `'wb'` depending on the input mode.

Lors de l'écriture d'un fichier, si la taille du fichier n'est pas connue mais peut être supérieure à 2 GiO, spécifiez `force_zip64=True` afin de vous assurer que le format d'en-tête est capable de supporter des fichiers volu-

mineux. Si la taille du fichier est connue à l'avance, instanciez un objet `ZipInfo` avec l'attribut `file_size` défini et utilisez-le en tant que paramètre *name*.

> **Note**
>
> Les méthodes `open()`, `read()` et `extract()` peuvent prendre un nom de fichier ou un objet `ZipInfo`. Cela est appréciable lorsqu'on essaie de lire un fichier ZIP qui contient des membres avec des noms en double.

Modifié dans la version 3.6 : Suppression de la gestion de `mode='U'`. Utilisez `io.TextIOWrapper` pour lire des fichiers texte compressés en mode *universal newlines*.

Modifié dans la version 3.6 : `ZipFile.open()` can now be used to write files into the archive with the `mode='w'` option.

Modifié dans la version 3.6 : Appeler `open()` sur un fichier ZipFile fermé lève une erreur `ValueError`. Précédemment, une erreur `RuntimeError` était levée.

Modifié dans la version 3.13 : Added attributes `name` and `mode` for the writeable file-like object. The value of the `mode` attribute for the readable file-like object was changed from `'r'` to `'rb'`.

`ZipFile.`**`extract`**(*member*, *path=None*, *pwd=None*)

Extract a member from the archive to the current working directory; *member* must be its full name or a `ZipInfo` object. Its file information is extracted as accurately as possible. *path* specifies a different directory to extract to. *member* can be a filename or a `ZipInfo` object. *pwd* is the password used for encrypted files as a `bytes` object.

Retourne le chemin normalisé créé (un dossier ou un nouveau fichier).

> **Note**
>
> Si le nom de fichier d'un membre est un chemin absolu, le disque/partage UNC et les (anti)slashes de départ seront supprimés, par exemple `///foo/bar` devient `foo/bar` sous Unix et `C:\foo\bar` devient `foo\bar` sous Windows. Et tous les composants `"..."` dans le nom de fichier d'un membre seront supprimés, par exemple `../../foo../../ba..r` devient `foo../ba..r`. Sous Windows les caractères illégaux (`:`, `<`, `>`, `|`, `"`, `?` et `*`) sont remplacés par un *underscore* (_).

Modifié dans la version 3.6 : Appeler `extract()` sur un fichier ZipFile fermé lève une erreur `ValueError`. Précédemment, une erreur `RuntimeError` était levée.

Modifié dans la version 3.6.2 : Le paramètre *path* accepte un objet chemin-compatible *path-like object*.

`ZipFile.`**`extractall`**(*path=None*, *members=None*, *pwd=None*)

Extract all members from the archive to the current working directory. *path* specifies a different directory to extract to. *members* is optional and must be a subset of the list returned by `namelist()`. *pwd* is the password used for encrypted files as a `bytes` object.

> **Avertissement**
>
> N'extrayez jamais d'archives depuis des sources non fiables sans inspection préalable. Il est possible que des fichiers soient créés en dehors de *path*, par exemple des membres qui ont des chemins de fichier absolus commençant par `"/"` ou des noms de fichier avec deux points `"..."`. Ce module essaie de prévenir ceci. Voir la note de `extract()`.

Modifié dans la version 3.6 : Appeler `extractall()` sur un fichier ZipFile fermé lève une erreur `ValueError`. Précédemment, une erreur `RuntimeError` était levée.

Modifié dans la version 3.6.2 : Le paramètre *path* accepte un objet chemin-compatible *path-like object*.

`ZipFile.printdir()`

Affiche la liste des contenus de l'archive sur `sys.stdout`.

`ZipFile.setpassword(pwd)`

Set *pwd* (a `bytes` object) as default password to extract encrypted files.

`ZipFile.read(name, pwd=None)`

Return the bytes of the file *name* in the archive. *name* is the name of the file in the archive, or a `ZipInfo` object. The archive must be open for read or append. *pwd* is the password used for encrypted files as a `bytes` object and, if specified, overrides the default password set with `setpassword()`. Calling `read()` on a ZipFile that uses a compression method other than `ZIP_STORED`, `ZIP_DEFLATED`, `ZIP_BZIP2` or `ZIP_LZMA` will raise a `NotImplementedError`. An error will also be raised if the corresponding compression module is not available.

Modifié dans la version 3.6 : Appeler `read()` sur un fichier ZipFile fermé lève une erreur `ValueError`. Précédemment, une erreur `RuntimeError` était levée.

`ZipFile.testzip()`

Lit tous les fichiers de l'archive et vérifie leurs sommes CRC et leurs en-têtes. Retourne le nom du premier fichier mauvais ou retourne `None` sinon.

Modifié dans la version 3.6 : Appeler `testzip()` sur un fichier ZipFile fermé lève une erreur `ValueError`. Précédemment, une erreur `RuntimeError` était levée.

`ZipFile.write(filename, arcname=None, compress_type=None, compresslevel=None)`

Écrit le fichier nommé *filename* dans l'archive, lui donnant *arcname* comme nom dans l'archive (par défaut, *arcname* prend la même valeur que *filename* mais sans lettre de disque et séparateur de chemin en première position). Si donné, *compress_type* écrase la valeur donnée pour le paramètre *compression* au constructeur pour la nouvelle entrée. De la même manière, *compression* écrase le constructeur si donné. L'archive doit être ouverte avec le mode `'w'`, `'x'` ou `'a'`.

> **Note**
>
> The ZIP file standard historically did not specify a metadata encoding, but strongly recommended CP437 (the original IBM PC encoding) for interoperability. Recent versions allow use of UTF-8 (only). In this module, UTF-8 will automatically be used to write the member names if they contain any non-ASCII characters. It is not possible to write member names in any encoding other than ASCII or UTF-8.

> **Note**
>
> Les noms d'archive doivent être relatifs à la racine de l'archive, c'est-à-dire qu'ils ne doivent pas commencer par un séparateur de chemin.

> **Note**
>
> Si `arcname` (ou `filename` si `arcname` n'est pas donné) contient un octet nul, le nom du fichier dans l'archive sera tronqué à l'octet nul.

> **Note**
>
> A leading slash in the filename may lead to the archive being impossible to open in some zip programs on Windows systems.

Modifié dans la version 3.6 : Appeler `write()` sur un fichier ZipFile fermé lève une erreur `ValueError`. Précédemment, une erreur `RuntimeError` était levée.

`ZipFile.writestr(zinfo_or_arcname, data, compress_type=None, compresslevel=None)`

Écrit un fichier dans l'archive. Le contenu est *data*, qui peut être soit une instance de `str` ou une instance de `bytes`; s'il s'agit d'une `str`, il est encodé en UTF-8 au préalable. *zinfo_or_arcname* est soit le nom de fichier qu'il sera donné dans l'archive, soit une instance de `ZipInfo`. Si c'est une instance, au moins le nom de fichier, la date et l'heure doivent être donnés. S'il s'agit d'un nom, la date et l'heure sont définies sur la date et l'heure actuelles. L'archive doit être ouverte avec le mode `'w'`, `'x'` ou `'a'`.

Si donné, *compress_type* écrase la valeur donnée pour le paramètre *compression* au constructeur de la nouvelle entrée ou dans le paramètre *zinfo_or_arcname* (si c'est une instance de `ZipInfo`). De la même manière, *compresslevel* le constructeur si donné.

> **Note**
>
> Lorsque l'on passe une instance de `ZipInfo` dans le paramètre *zinfo_or_arcname*, la méthode de compression utilisée sera celle spécifiée dans le membre *compress_type* de l'instance `ZipInfo` donnée. Par défaut, le constructeur de la classe `ZipInfo` définit ce membre à `ZIP_STORED`.

Modifié dans la version 3.2 : L'argument *compress_type*.

Modifié dans la version 3.6 : Appeler `writestr()` sur un fichier ZipFile fermé lève une erreur `ValueError`. Précédemment, une erreur `RuntimeError` était levée.

`ZipFile.mkdir(zinfo_or_directory, mode=511)`

Create a directory inside the archive. If *zinfo_or_directory* is a string, a directory is created inside the archive with the mode that is specified in the *mode* argument. If, however, *zinfo_or_directory* is a `ZipInfo` instance then the *mode* argument is ignored.

The archive must be opened with mode `'w'`, `'x'` or `'a'`.

Ajouté dans la version 3.11.

Les attributs suivants sont aussi disponibles :

`ZipFile.filename`

Nom du fichier ZIP.

`ZipFile.debug`

Le niveau d'affichage de *debug* à utiliser. Peut être défini de `0` (par défaut, pas d'affichage) à `3` (affichage le plus bavard). Les informations de débogage sont affichées sur `sys.stdout`.

`ZipFile.comment`

Le commentaire associé au fichier ZIP en tant qu'objet `bytes`. Si vous affectez un commentaire à une instance de `ZipFile` créée avec le mode `'w'`, `'x'` ou `'a'`, il ne doit pas dépasser 65535 octets. Les commentaires plus longs que cette taille seront tronqués.

13.5.2 Objets *Path*

class `zipfile.Path(root, at='')`

Construit un objet *Path* depuis le fichier ZIP *root* (qui peut être une instance de `ZipFile` ou tout ce qui sera accepté par le paramètre *file* du constructeur de `ZipFile`).

`at` indique la position de ce *Path* dans l'archive ZIP, par exemple `"dir/file.txt"`, `"dir/"`, ou `""`. Par défaut c'est une chaine vide, indiquant la racine de l'archive.

Les objets `Path` de `zipfile` exposent les fonctionnalités suivantes des objets de `pathlib.Path` :

Path objects are traversable using the `/` operator or `joinpath`.

`Path.name`

Le dernier segment du chemin.

`Path.open(mode='r', *, pwd, **)`

Appelle `ZipFile.open()` sur ce chemin. Il est possible d'ouvrir en lecture ou en écriture, au format texte ou binaire, via les modes `'r'`, `'w'`, `'rb'` et `'wb'`. Les arguments positionnels et nommés sont passés à la classe `io.TextIOWrapper` lors d'une ouverture au format texte, ils sont ignorés autrement. `pwd` est le paramètre `pwd` de `ZipFile.open()`.

Modifié dans la version 3.9 : prise en charge des modes texte et binaire pour `open`. Le mode texte est maintenant le mode par défaut.

Modifié dans la version 3.11.2 : The `encoding` parameter can be supplied as a positional argument without causing a `TypeError`. As it could in 3.9. Code needing to be compatible with unpatched 3.10 and 3.11 versions must pass all `io.TextIOWrapper` arguments, `encoding` included, as keywords.

`Path.iterdir()`

Énumère le contenu du dossier actuel.

`Path.is_dir()`

Renvoie `True` si ce chemin pointe sur un dossier.

`Path.is_file()`

Renvoie `True` si ce chemin pointe sur un fichier.

`Path.is_symlink()`

Return `True` if the current context references a symbolic link.

Ajouté dans la version 3.12.

Modifié dans la version 3.13 : Previously, `is_symlink` would unconditionally return `False`.

`Path.exists()`

Renvoie `True` si le chemin pointe sur un fichier ou un dossier de l'archive ZIP

`Path.suffix`

The last dot-separated portion of the final component, if any. This is commonly called the file extension.

Ajouté dans la version 3.11 : Added `Path.suffix` property.

`Path.stem`

The final path component, without its suffix.

Ajouté dans la version 3.11 : Added `Path.stem` property.

`Path.suffixes`

A list of the path's suffixes, commonly called file extensions.

Ajouté dans la version 3.11 : Added `Path.suffixes` property.

`Path.read_text(*, **)`

Lit le fichier au format texte. Les arguments positionnels et nommés sont passés à `io.TextIOWrapper` (sauf `buffer`, qui est imposé par le contexte)

Modifié dans la version 3.11.2 : The `encoding` parameter can be supplied as a positional argument without causing a `TypeError`. As it could in 3.9. Code needing to be compatible with unpatched 3.10 and 3.11 versions must pass all `io.TextIOWrapper` arguments, `encoding` included, as keywords.

`Path.read_bytes()`

Lit le fichier en mode binaire, renvoyant un objet *bytes*.

`Path.joinpath(*other)`

Return a new Path object with each of the *other* arguments joined. The following are equivalent :

```
>>> Path(...).joinpath('child').joinpath('grandchild')
>>> Path(...).joinpath('child', 'grandchild')
>>> Path(...) / 'child' / 'grandchild'
```

Modifié dans la version 3.10 : Prior to 3.10, `joinpath` was undocumented and accepted exactly one parameter.

The zipp project provides backports of the latest path object functionality to older Pythons. Use `zipp.Path` in place of `zipfile.Path` for early access to changes.

13.5.3 Objets *PyZipFile*

Le constructeur de `PyZipFile` prend les mêmes paramètres que le constructeur de `ZipFile` avec un paramètre additionnel *optimize*.

class zipfile.**PyZipFile**(*file*, *mode='r'*, *compression=ZIP_STORED*, *allowZip64=True*, *optimize=-1*)

Modifié dans la version 3.2 : Added the *optimize* parameter.

Modifié dans la version 3.4 : Les extensions ZIP64 sont activées par défaut.

Les instances ont une méthode supplémentaire par rapport aux objets `ZipFile` :

writepy(*pathname*, *basename=''*, *filterfunc=None*)

Cherche les fichiers `*.py` et ajoute le fichier correspondant à l'archive.

Si le paramètre *optimize* du constructeur de `PyZipFile` n'a pas été donné ou est à `-1`, le fichier correspondant est un fichier `*.pyc`, à compiler si nécessaire.

Si le paramètre *optimize* du constructeur de `PyZipFile` est à `0`, `1` ou `2`, ne sont ajoutés dans l'archive que les fichiers avec ce niveau d'optimisation (voir `compile()`), à compiler si nécessaire.

Si *pathname* est un fichier, le chemin de fichier doit terminer par `.py` et uniquement le fichier (`*.pyc` correspondant) est ajouté au niveau le plus haut (sans information de chemin). Si *pathname* est un fichier ne terminant pas par `.py`, une exception `RuntimeError` est levée. Si c'est un répertoire et que le répertoire n'est pas un répertoire de paquet, alors tous les fichiers `*.pyc` sont ajoutés à la racine. Si le répertoire est un répertoire de paquet, alors tous les `*.pyc` sont ajoutés sous le nom du paquet en tant que chemin, et s'il y a des sous-répertoires qui sont des répertoires de paquet, ils sont tous ajoutés récursivement dans un ordre trié.

basename n'est sensé être utilisé qu'en interne.

filterfunc, si donné, doit être une fonction prenant une seule chaîne de caractères en argument. Il lui sera passé chaque chemin (incluant chaque chemin de fichier complet individuel) avant d'être ajouté à l'archive. Si *filterfunc* retourne une valeur fausse, le chemin n'est pas ajouté et si c'est un répertoire son contenu est ignoré. Par exemple, si nos fichiers de test sont tous soit dans des répertoires `test` ou commencent par `test_`, nous pouvons utiliser une fonction *filterfunc* pour les exclure :

```
>>> zf = PyZipFile('myprog.zip')
>>> def notests(s):
...     fn = os.path.basename(s)
...     return (not (fn == 'test' or fn.startswith('test_')))
...
>>> zf.writepy('myprog', filterfunc=notests)
```

La méthode `writepy()` crée des archives avec des noms de fichier comme suit :

```
string.pyc                    # Top level name
test/__init__.pyc             # Package directory
test/testall.pyc              # Module test.testall
test/bogus/__init__.pyc       # Subpackage directory
test/bogus/myfile.pyc         # Submodule test.bogus.myfile
```

Modifié dans la version 3.4 : Added the *filterfunc* parameter.

Modifié dans la version 3.6.2 : Le paramètre *pathname* accepte un objet chemin-compatible *path-like object*.

Modifié dans la version 3.7 : La récursion trie les entrées de dossier.

13.5.4 Objets *ZipInfo*

Des instances de la classe `ZipInfo` sont retournées par les méthodes `getinfo()` et `infolist()` des objets `ZipFile`. Chaque objet stocke des informations sur un seul membre de l'archive ZIP.

Il y a une méthode de classe pour créer une instance de `ZipInfo` pour un fichier du système de fichiers :

classmethod `ZipInfo.`**`from_file`**(*filename*, *arcname=None*, *, *strict_timestamps=True*)

Construit une instance de `ZipInfo` pour le fichier du système de fichiers, en préparation de l'ajouter à un fichier ZIP.

filename doit être un chemin vers un fichier ou un répertoire dans le système de fichiers.

Si *arcname* est spécifié, il est utilisé en tant que nom dans l'archive. Si *arcname* n'est pas spécifié, le nom sera le même que *filename* mais sans lettre de disque et sans séparateur de chemin en première position.

Les fichiers ZIP plus anciens que le 1er janvier 1980 sont autorisés lorsque l'argument *strict_timestamps* vaut `False`, moyennant de les voir datés du 1er janvier 1980. De même pour les fichiers datés d'après le 31 décembre 2107 qui voient leur horodatage fixé au 31 décembre 2107.

Ajouté dans la version 3.6.

Modifié dans la version 3.6.2 : Le paramètre *filename* accepte un objet chemin-compatible *path-like object*.

Modifié dans la version 3.8 : Added the *strict_timestamps* keyword-only parameter.

Les instances ont les méthodes et attributs suivants :

`ZipInfo.`**`is_dir`**`()`

Retourne `True` si le membre d'archive est un répertoire.

Utilise le nom de l'entrée : les répertoires doivent toujours se terminer par `/`.

Ajouté dans la version 3.6.

`ZipInfo.`**`filename`**

Nom du fichier dans l'archive.

`ZipInfo.`**`date_time`**

Date et heure de dernière modification pour le membre de l'archive. *Tuple* de six valeurs :

Index	Valeur
0	Année (>= 1980)
1	Mois (indexé à partir de 1)
2	Jour du mois (indexé à partir de 1)
3	Heures (indexées à partir de 0)
4	Minutes (indexées à partir de 0)
5	Secondes (indexées à partir de 0)

> **Note**
>
> Le format de fichier ZIP ne gère pas les horodatages avant 1980.

`ZipInfo.`**`compress_type`**

Type de compression du membre d'archive.

`ZipInfo.`**`comment`**

Commentaire pour le membre d'archive individuel en tant qu'objet `bytes`.

`ZipInfo.`**`extra`**

Données du champ d'extension. La documentation PKZIP Application Note contient quelques commentaires sur la structure interne des données contenues dans cet objet `bytes`.

`ZipInfo.create_system`
Système ayant créé l'archive ZIP.

`ZipInfo.create_version`
Version de PKZIP ayant créé l'archive ZIP.

`ZipInfo.extract_version`
Version de PKZIP nécessaire à l'extraction de l'archive ZIP.

`ZipInfo.reserved`
Doit être à zéro.

`ZipInfo.flag_bits`
Bits d'options ZIP.

`ZipInfo.volume`
Numéro de volume de l'entête du fichier.

`ZipInfo.internal_attr`
Attributs internes.

`ZipInfo.external_attr`
Attributs de fichier externes.

`ZipInfo.header_offset`
Longueur de l'entête du fichier en octets.

`ZipInfo.CRC`
CRC-32 du fichier décompressé.

`ZipInfo.compress_size`
Taille des données décompressées.

`ZipInfo.file_size`
Taille du fichier décompressé.

13.5.5 Interface en ligne de commande

Le module `zipfile` fournit une interface en ligne de commande simple pour interagir avec des archives ZIP.

Si vous voulez créer une nouvelle archive ZIP, spécifiez son nom après l'option `-c` et listez ensuite le(s) nom(s) de fichier à inclure :

```
$ python -m zipfile -c monty.zip spam.txt eggs.txt
```

Passer un répertoire est aussi possible :

```
$ python -m zipfile -c monty.zip life-of-brian_1979/
```

Si vous voulez extraire une archive ZIP dans un répertoire donné, utilisez l'option `-e` :

```
$ python -m zipfile -e monty.zip target-dir/
```

Pour une liste des fichiers dans une archive ZIP, utilisez l'option `-l` :

```
$ python -m zipfile -l monty.zip
```

Options de la ligne de commande

-l <zipfile>

--list <zipfile>

Liste les fichiers dans un fichier ZIP *zipfile*.

-c <zipfile> <source1> ... <sourceN>

--create <zipfile> <source1> ... <sourceN>

Crée un fichier ZIP *zipfile* à partir des fichiers *source*.

-e <zipfile> <output_dir>

--extract <zipfile> <output_dir>

Extrait le fichier ZIP *zipfile* vers le répertoire cible *output_dir*.

-t <zipfile>

--test <zipfile>

Teste si le fichier zip est valide.

--metadata-encoding <encoding>

Specify encoding of member names for `-l`, `-e` and `-t`.

Ajouté dans la version 3.11.

13.5.6 Problèmes de décompression

L'extraction d'une archive ZIP par le module *zipfile* peut échouer pour les raisons listées ci-dessous.

À cause du fichier lui-même

La décompression peut échouer à cause d'un mot de passe ou d'une somme de contrôle CRC incorrectes. Elle peut aussi échouer si le format, la méthode de compression, ou de chiffrement n'est pas implémenté.

Limitations du système de fichiers

Dépasser les limites du système de fichiers peut faire échouer la décompression. Ces limites peuvent concerner les caractères licites pour un nom de fichier, la longueur du nom du fichier ou du chemin, la taille d'un fichier, le nombre de fichiers, etc.

Ressources limitées

Le manque de mémoire ou d'espace disque peut mener à un échec de décompression. Par exemple, une bombe de décompression (ZIP bomb), décompressés avec *zifile* peut remplir l'espace disque.

Interruption

Une interruption durant la décompression, en utilisant *control-C* ou en tuant le processus, peut mener à une décompression partielle de l'archive.

Comportements par défaut de l'extraction

Ne pas connaître le comportement d'extraction par défaut peut causer des résultats inattendus. Par exemple, lors de l'extraction d'une même archive deux fois, les fichiers sont écrasés sans prévenir.

13.6 `tarfile` --- Read and write tar archive files

Code source : Lib/tarfile.py

Le module `tarfile` rend possible la lecture et l'écriture des archives *tar*, incluant celles utilisant la compression *gzip*, *bz2* et *lzma*. Utilisez le module `zipfile` pour lire ou écrire des fichiers `zip`, ou les fonctions de niveau supérieur dans *shutil*.

Quelques faits et chiffres :

— lit et écrit des archives compressées avec `gzip`, `bz2` ou `lzma` si les modules respectifs sont disponibles.
— prise en charge de la lecture/écriture pour le format *POSIX.1-1988* (*ustar*).
— prise en charge de la lecture/écriture pour le format GNU *tar* incluant les extensions *longname* et *longlink*, prise en charge de la lecture seule de toutes les variantes de l'extension *sparse* incluant la restauration des fichiers discontinus.
— prise en charge de la lecture/écriture pour le format *POSIX.1-2001* (*pax*).
— gère les répertoires, les fichiers normaux, les liens directs (*hard links* en anglais), les liens symboliques, les tubes nommés (*FIFO* en anglais), les périphériques de caractère et les périphériques de bloc et est en mesure d'acquérir et de restaurer les informations du fichier comme l'horodatage, les autorisations d'accès et le propriétaire.

Modifié dans la version 3.3 : prise en charge de la compression `lzma`.

Modifié dans la version 3.12 : Archives are extracted using a *filter*, which makes it possible to either limit surprising/dangerous features, or to acknowledge that they are expected and the archive is fully trusted.

Modifié dans la version 3.14 : Set the default extraction filter to `data`, which disallows some dangerous features such as links to absolute paths or paths outside of the destination. Previously, the filter strategy was equivalent to `fully_trusted`.

`tarfile.`**`open`**(*name=None*, *mode='r'*, *fileobj=None*, *bufsize=10240*, ***kwargs*)

Renvoie un objet `TarFile` pour le nom de chemin *name*. Pour plus d'informations sur les objets `TarFile` et les mot-clefs arguments permis, voir *Les objets TarFile*.

Le *mode* doit être une chaîne de caractères de la forme `'filemode[:compression]'`, par défaut à `'r'`. Voici une liste complète des combinaisons de mode :

mode	action
`'r'` ou `'r:*'`	Ouvre en lecture avec compression transparente (recommandé).
`'r:'`	Ouvre en lecture, sans compression.
`'r:gz'`	Ouvre en lecture avec la compression *gzip*.
`'r:bz2'`	Ouvre en lecture avec la compression *bzip2*.
`'r:xz'`	Ouvre en lecture avec la compression *lzma*.
`'x'` ou `'x:'`	Create a tarfile exclusively without compression. Raise a `FileExistsError` exception if it already exists.
`'x:gz'`	Create a tarfile with gzip compression. Raise a `FileExistsError` exception if it already exists.
`'x:bz2'`	Create a tarfile with bzip2 compression. Raise a `FileExistsError` exception if it already exists.
`'x:xz'`	Create a tarfile with lzma compression. Raise a `FileExistsError` exception if it already exists.
`'a'` ou `'a:'`	Ouvre pour ajouter à la fin, sans compression. Le fichier est créé s'il n'existe pas.
`'w'` ou `'w:'`	Ouvre en écriture, sans compression.
`'w:gz'`	Ouvre en écriture avec compression *gzip*.
`'w:bz2'`	Ouvre en écriture avec compression *bzip2*.
`'w:xz'`	Ouvre en écriture avec la compression *lzma*.

Notez que les combinaisons `'a:gz'`, `'a:bz2'` ou `'a:xz'` ne sont pas possibles. Si le mode n'est pas adapté pour ouvrir un certain fichier (compressé) pour la lecture, une exception `ReadError` est levée. Utilisez le mode `'r'` pour éviter cela. Si une méthode de compression n'est pas prise en charge, `CompressionError` est levée.

Si *fileobj* est spécifié, il est utilisé comme une alternative au *file object* ouvert en mode binaire pour *name*. Il est censé être à la position 0.

For modes `'w:gz'`, `'x:gz'`, `'w|gz'`, `'w:bz2'`, `'x:bz2'`, `'w|bz2'`, `tarfile.open()` accepts the keyword argument *compresslevel* (default `9`) to specify the compression level of the file.

Pour les modes `'w:xz'` et `'x:xz'`, `tarfile.open()` accepte l'argument nommé *preset* pour spécifier le niveau de compression du fichier.

For special purposes, there is a second format for *mode* : `'filemode|[compression]'`. `tarfile.open()` will return a `TarFile` object that processes its data as a stream of blocks. No random seeking will be done on the file. If given, *fileobj* may be any object that has a `read()` or `write()` method (depending on the *mode*) that works with bytes. *bufsize* specifies the blocksize and defaults to `20 * 512` bytes. Use this variant in combination with e.g. `sys.stdin.buffer`, a socket *file object* or a tape device. However, such a `TarFile` object is limited in that it does not allow random access, see *Exemples*. The currently possible modes :

Mode	Action
`'r|*'`	Ouvre un *flux* des blocs de *tar* en lecture avec une compression transparente.
`'r|'`	Ouvre un *flux* de blocs *tar* non compressés en lecture.
`'r|gz'`	Ouvre un flux compressé avec *gzip* en lecture.
`'r|bz2'`	Ouvre un *flux* compressé avec *bzip2* en lecture.
`'r|xz'`	Ouvre un *flux* compressé avec *lzma* en lecture.
`'w|'`	Ouvre un *flux* non compressé en écriture.
`'w|gz'`	Ouvre un *flux* compressé avec *gzip* en écriture.
`'w|bz2'`	Ouvre un *flux* compressé avec *bzip2* en écriture.
`'w|xz'`	Ouvre un *flux* compressé avec *lzma* en écriture.

Modifié dans la version 3.5 : le mode `'x'` (création exclusive) a été ajouté.

Modifié dans la version 3.6 : le paramètre *name* accepte un *path-like object*.

Modifié dans la version 3.12 : The *compresslevel* keyword argument also works for streams.

class tarfile.**TarFile**

Classe pour la lecture et l'écriture d'archives *tar*. N'utilisez pas cette classe directement, préférez `tarfile.open()`. Voir *Les objets TarFile*.

tarfile.**is_tarfile**(*name*)

Renvoie `True` si *name* est un fichier d'archive *tar*, que le module `tarfile` peut lire. *name* peut être une `str`, un ficher ou un objet fichier-compatible.

Modifié dans la version 3.9 : Prise en charge des fichiers et des objets fichier-compatibles.

Le module `tarfile` définit les exceptions suivantes :

exception tarfile.**TarError**

Classe de base pour toutes les exceptions du module `tarfile`.

exception tarfile.**ReadError**

Est levée lors de l'ouverture d'une archive *tar*, qui ne peut pas être gérée par le module `tarfile` ou est invalide.

exception tarfile.**CompressionError**

Est levée lorsqu'une méthode de compression n'est pas prise en charge ou lorsque les données ne peuvent pas être décodées correctement.

exception tarfile.**StreamError**

Est levée pour les limitations typiques des objets de type flux `TarFile`.

exception tarfile.**ExtractError**

Est levée pour des erreurs *non-fatales* lors de l'utilisation de `TarFile.extract()`, mais uniquement si `TarFile.errorlevel== 2`.

exception tarfile.**HeaderError**

Est levée par `TarInfo.frombuf()` si le tampon qu'il obtient n'est pas valide.

exception tarfile.**FilterError**

Base class for members *refused* by filters.

tarinfo
Information about the member that the filter refused to extract, as *TarInfo*.

exception tarfile.**AbsolutePathError**
Raised to refuse extracting a member with an absolute path.

exception tarfile.**OutsideDestinationError**
Raised to refuse extracting a member outside the destination directory.

exception tarfile.**SpecialFileError**
Raised to refuse extracting a special file (e.g. a device or pipe).

exception tarfile.**AbsoluteLinkError**
Raised to refuse extracting a symbolic link with an absolute path.

exception tarfile.**LinkOutsideDestinationError**
Raised to refuse extracting a symbolic link pointing outside the destination directory.

Les constantes suivantes sont disponibles au niveau du module :

tarfile.**ENCODING**
L'encodage des caractères par défaut est `'utf-8'` sous Windows, sinon la valeur renvoyée par `sys.getfilesystemencoding()`.

tarfile.**REGTYPE**
tarfile.**AREGTYPE**
A regular file `type`.

tarfile.**LNKTYPE**
A link (inside tarfile) `type`.

tarfile.**SYMTYPE**
A symbolic link `type`.

tarfile.**CHRTYPE**
A character special device `type`.

tarfile.**BLKTYPE**
A block special device `type`.

tarfile.**DIRTYPE**
A directory `type`.

tarfile.**FIFOTYPE**
A FIFO special device `type`.

tarfile.**CONTTYPE**
A contiguous file `type`.

tarfile.**GNUTYPE_LONGNAME**
A GNU tar longname `type`.

tarfile.**GNUTYPE_LONGLINK**
A GNU tar longlink `type`.

tarfile.**GNUTYPE_SPARSE**
A GNU tar sparse file `type`.

Chacune des constantes suivantes définit un format d'archive *tar* que le module `tarfile` est capable de créer. Voir la section *Formats tar pris en charge* pour plus de détails.

tarfile.**USTAR_FORMAT**
Le format *POSIX.1-1988* (*ustar*).

`tarfile.GNU_FORMAT`

Le format GNU *tar*.

`tarfile.PAX_FORMAT`

Le format *POSIX.1-2001* (*pax*).

`tarfile.DEFAULT_FORMAT`

Format par défaut pour la création d'archives. C'est actuellement `PAX_FORMAT`.

Modifié dans la version 3.8 : Le format par défaut des nouvelles archives a été changé de `GNU_FORMAT` en `PAX_FORMAT`.

Voir aussi

Module `zipfile`
Documentation du module standard `zipfile`.

Archiving operations
Documentation des outils d'archivage de haut niveau fournis par le module standard `shutil`.

Manuel GNU *tar*, format *tar* basique (en anglais)
Documentation pour les fichiers d'archive *tar*, y compris les extensions *tar* GNU.

13.6.1 Les objets *TarFile*

L'objet `TarFile` fournit une interface vers une archive *tar*. Une archive *tar* est une séquence de blocs. Un membre d'archive (un fichier stocké) est composé d'un bloc d'en-tête suivi des blocs de données. Il est possible de stocker plusieurs fois un fichier dans une archive *tar*. Chaque membre d'archive est représenté par un objet `TarInfo`, voir *Les objets TarInfo* pour plus de détails.

Un objet `TarFile` peut être utilisé comme gestionnaire de contexte dans une instruction `with`. Il sera automatiquement fermé une fois le bloc terminé. Veuillez noter qu'en cas d'exception, une archive ouverte en écriture ne sera pas finalisée ; seul l'objet fichier utilisé en interne sera fermé. Voir la section *Exemples* pour un cas d'utilisation.

Ajouté dans la version 3.2 : Ajout de la prise en charge du protocole de gestion de contexte.

class `tarfile.TarFile`(*name=None*, *mode='r'*, *fileobj=None*, *format=DEFAULT_FORMAT*, *tarinfo=TarInfo*, *dereference=False*, *ignore_zeros=False*, *encoding=ENCODING*, *errors='surrogateescape'*, *pax_headers=None*, *debug=0*, *errorlevel=1*, *stream=False*)

Tous les arguments suivants sont facultatifs et sont également accessibles en tant qu'instance d'attributs.

name is the pathname of the archive. *name* may be a *path-like object*. It can be omitted if *fileobj* is given. In this case, the file object's `name` attribute is used if it exists.

Le *mode* est soit `'r'` pour lire à partir d'une archive existante, `'a'` pour ajouter des données à un fichier existant, `'w'` pour créer un nouveau fichier en écrasant un existant, ou `'x'` pour créer un nouveau fichier uniquement s'il n'existe pas déjà.

Si *fileobj* est fourni, il est utilisé pour lire ou écrire des données. S'il peut être déterminé, le *mode* est remplacé par le mode de *fileobj*. *fileobj* sera utilisé à partir de la position 0.

Note

fileobj n'est pas fermé, lorsque `TarFile` est fermé.

Le *format* contrôle le format d'archive en écriture. Il doit s'agir de l'une des constantes `USTAR_FORMAT`, `GNU_FORMAT` ou `PAX_FORMAT` définies au niveau du module. Lors de la lecture, le format sera automatiquement détecté, même si différents formats sont présents dans une même archive.

L'argument *tarinfo* peut être utilisé pour remplacer la classe par défaut `TarInfo` par une autre.

Si *dereference* est `False`, ajoute des liens symboliques et physiques à l'archive. Si c'est `True`, ajoute le contenu des fichiers cibles à l'archive. Cela n'a aucun effet sur les systèmes qui ne prennent pas en charge les liens symboliques.

Si *ignore_zeros* est `False`, traite un bloc vide comme la fin de l'archive. Si c'est le cas `True`, saute les blocs vides (et invalides) et essaye d'obtenir autant de membres que possible. Ceci n'est utile que pour lire des archives concaténées ou endommagées.

debug peut être défini de `0` (aucun message de débogage) à `3` (tous les messages de débogage). Les messages sont écrits dans `sys.stderr`.

errorlevel controls how extraction errors are handled, see *the corresponding attribute*.

Les arguments *encoding* et *errors* définissent l'encodage de caractères à utiliser pour lire ou écrire l'archive et comment les erreurs de conversion vont être traitées. Les paramètres par défaut fonctionneront pour la plupart des utilisateurs. Voir la section *Problèmes unicode* pour des informations détaillées.

L'argument *pax_headers* est un dictionnaire facultatif de chaînes de caractères qui sera ajouté en tant qu'en-tête global *pax* si le *format* est `PAX_FORMAT`.

If *stream* is set to `True` then while reading the archive info about files in the archive are not cached, saving memory.

Modifié dans la version 3.2 : Utilise `'surrogateescape'` comme valeur par défaut pour l'argument *errors*.

Modifié dans la version 3.5 : le mode `'x'` (création exclusive) a été ajouté.

Modifié dans la version 3.6 : le paramètre *name* accepte un *path-like object*.

Modifié dans la version 3.13 : Add the *stream* parameter.

classmethod `TarFile.`**`open`**`(...)`

Constructeur alternatif. La fonction `tarfile.open()` est en fait un raccourci vers cette méthode de classe.

`TarFile.`**`getmember`**`(`*name*`)`

Renvoie un objet `TarInfo` pour le membre *name*. Si *name* est introuvable dans l'archive, `KeyError` est levée.

> **Note**
>
> Si un membre apparaît plus d'une fois dans l'archive, sa dernière occurrence est supposée être la version la plus récente.

`TarFile.`**`getmembers`**`()`

Renvoie les membres de l'archive sous la forme d'une liste d'objets `TarInfo`. La liste a le même ordre que les membres de l'archive.

`TarFile.`**`getnames`**`()`

Renvoie les membres comme une liste de leurs noms. Il a le même ordre que la liste renvoyée par `getmembers()`.

`TarFile.`**`list`**`(`*verbose=True*, *, *members=None*`)`

Imprime une table des matières dans `sys.stdout`. Si *verbose* est `False`, seuls les noms des membres sont imprimés. Si c'est `True`, une sortie similaire à celle de **`ls -l`** est produite. Si des *membres* facultatifs sont fournis, il doit s'agir d'un sous-ensemble de la liste renvoyée par `getmembers()`.

Modifié dans la version 3.5 : Ajout du paramètre *members*.

`TarFile.`**`next`**`()`

Renvoie le membre suivant de l'archive en tant qu'objet `TarInfo`, lorsque la classe `TarFile` est ouverte en lecture. Renvoie `None` s'il n'y a pas.

`TarFile.`**`extractall`**`(`*path='.'*, *members=None*, *, *numeric_owner=False*, *filter=None*`)`

Extrait tous les membres de l'archive vers le répertoire de travail actuel ou le répertoire *chemin*. Si des *membres*

facultatifs sont fournis, il doit s'agir d'un sous-ensemble de la liste renvoyée par `getmembers()`. Les informations d'annuaire telles que le propriétaire, l'heure de modification et les autorisations sont définies une fois tous les membres extraits. Cela est fait pour contourner deux problèmes : l'heure de modification d'un répertoire est réinitialisée chaque fois qu'un fichier y est créé. Et, si les autorisations d'un répertoire ne permettent pas l'écriture, l'extraction de fichiers échoue.

Si *numeric_owner* est `True`, les numéros *uid* et *gid* du fichier *tar* sont utilisés pour définir le propriétaire et le groupe des fichiers extraits. Sinon, les valeurs nommées du fichier *tar* sont utilisées.

The *filter* argument specifies how `members` are modified or rejected before extraction. See *Extraction filters* for details. It is recommended to set this explicitly only if specific *tar* features are required, or as `filter='data'` to support Python versions with a less secure default (3.13 and lower).

> **Avertissement**
>
> Never extract archives from untrusted sources without prior inspection.
>
> Since Python 3.14, the default (`data`) will prevent the most dangerous security issues. However, it will not prevent *all* unintended or insecure behavior. Read the *Extraction filters* section for details.

Modifié dans la version 3.5 : Ajout du paramètre *numeric_owner*.

Modifié dans la version 3.6 : Le paramètre *path* accepte un *path-like object*.

Modifié dans la version 3.12 : Ajout du paramètre *filter*.

Modifié dans la version 3.14 : The *filter* parameter now defaults to `'data'`.

`TarFile.`**`extract`**(*member*, *path=''*, *set_attrs=True*, **, numeric_owner=False*, *filter=None*)

Extrait un membre de l'archive vers le répertoire de travail actuel, en utilisant son nom complet. Les informations de son fichier sont extraites aussi précisément que possible. Le membre peut être un nom de fichier ou un objet `TarInfo`. Vous pouvez spécifier un répertoire différent en utilisant *path*. *path* peut être un *path-like object*. Les attributs de fichier (propriétaire, *mtime*, mode) sont définis sauf si *set_attrs* est faux.

The *numeric_owner* and *filter* arguments are the same as for `extractall()`.

> **Note**
>
> La méthode `extract()` ne prend pas en charge plusieurs problèmes d'extraction. Dans la plupart des cas, vous devriez envisager d'utiliser la méthode `extractall()`.

> **Avertissement**
>
> Never extract archives from untrusted sources without prior inspection. See the warning for `extractall()` for details.

Modifié dans la version 3.2 : Ajout du paramètre *set_attrs*.

Modifié dans la version 3.5 : Ajout du paramètre *numeric_owner*.

Modifié dans la version 3.6 : Le paramètre *path* accepte un *path-like object*.

Modifié dans la version 3.12 : Ajout du paramètre *filter*.

`TarFile.`**`extractfile`**(*member*)

Extrait un membre de l'archive en tant qu'objet fichier. *member* peut être un nom de fichier ou un objet `TarInfo`. Si *member* est un fichier normal ou un lien, un objet `io.BufferedReader` est renvoyé. Sinon, `None` est renvoyé. Lève une exception `KeyError` si *member* n'est pas présent dans l'archive.

Modifié dans la version 3.3 : Renvoie un objet `io.BufferedReader`.

Modifié dans la version 3.13 : The returned `io.BufferedReader` object has the `mode` attribute which is always equal to `'rb'`.

`TarFile.errorlevel: int`

If *errorlevel* is `0`, errors are ignored when using `TarFile.extract()` and `TarFile.extractall()`. Nevertheless, they appear as error messages in the debug output when *debug* is greater than 0. If `1` (the default), all *fatal* errors are raised as `OSError` or `FilterError` exceptions. If `2`, all *non-fatal* errors are raised as `TarError` exceptions as well.

Some exceptions, e.g. ones caused by wrong argument types or data corruption, are always raised.

Custom *extraction filters* should raise `FilterError` for *fatal* errors and `ExtractError` for *non-fatal* ones.

Note that when an exception is raised, the archive may be partially extracted. It is the user's responsibility to clean up.

`TarFile.extraction_filter`

Ajouté dans la version 3.12.

The *extraction filter* used as a default for the *filter* argument of `extract()` and `extractall()`.

The attribute may be `None` or a callable. String names are not allowed for this attribute, unlike the *filter* argument to `extract()`.

If `extraction_filter` is `None` (the default), extraction methods will use the `data` filter by default.

The attribute may be set on instances or overridden in subclasses. It also is possible to set it on the `TarFile` class itself to set a global default, although, since it affects all uses of *tarfile*, it is best practice to only do so in top-level applications or `site configuration`. To set a global default this way, a filter function needs to be wrapped in `staticmethod()` to prevent injection of a `self` argument.

Modifié dans la version 3.14 : The default filter is set to `data`, which disallows some dangerous features such as links to absolute paths or paths outside of the destination. Previously, the default was equivalent to `fully_trusted`.

`TarFile.add`(*name*, *arcname=None*, *recursive=True*, ***, *filter=None*)

Ajoute le fichier *name* à l'archive. *name* peut être n'importe quel type de fichier (répertoire, *fifo*, lien symbolique, etc.). S'il est donné, *arcname* spécifie un autre nom pour le fichier dans l'archive. Les répertoires sont ajoutés récursivement par défaut. Cela peut être évité en définissant *recursive* sur `False`. La récursivité ajoute des entrées dans l'ordre trié. Si *filter* est donné, il convient que ce soit une fonction qui prend un argument d'objet `TarInfo` et renvoie l'objet changé `TarInfo`. S'il renvoie à la place `None`, l'objet `TarInfo` sera exclu de l'archive. Voir *Exemples* pour un exemple.

Modifié dans la version 3.2 : Ajout du paramètre *filter*.

Modifié dans la version 3.7 : La récursivité ajoute les entrées dans un ordre trié.

`TarFile.addfile`(*tarinfo*, *fileobj=None*)

Add the `TarInfo` object *tarinfo* to the archive. If *tarinfo* represents a non zero-size regular file, the *fileobj* argument should be a *binary file*, and `tarinfo.size` bytes are read from it and added to the archive. You can create `TarInfo` objects directly, or by using `gettarinfo()`.

Modifié dans la version 3.13 : *fileobj* must be given for non-zero-sized regular files.

`TarFile.gettarinfo`(*name=None*, *arcname=None*, *fileobj=None*)

Crée un objet `TarInfo` à partir du résultat de `os.stat()` ou équivalent sur un fichier existant. Le fichier est soit nommé par *name*, soit spécifié comme *file object* *fileobj* avec un descripteur de fichier. *name* peut être un *objet* semblable à un chemin. S'il est donné, *arcname* spécifie un autre nom pour le fichier dans l'archive, sinon, le nom est tiré de l'attribut *fileobj* `name`, ou de l'argument *name*. Le nom doit être une chaîne de texte.

Vous pouvez modifier certains des attributs de `TarInfo` avant de les ajouter en utilisant `addfile()`. Si l'objet fichier n'est pas un objet fichier ordinaire positionné au début du fichier, des attributs tels que `size` peuvent nécessiter une modification. C'est le cas pour des objets tels que `GzipFile`. Le `name` peut également être modifié, auquel cas *arcname* pourrait être une chaîne factice.

Modifié dans la version 3.6 : le paramètre *name* accepte un *path-like object*.

`TarFile.close()`

Ferme le `TarFile`. En mode écriture, deux blocs de finition à zéro sont ajoutés à l'archive.

`TarFile.pax_headers: dict`

Un dictionnaire contenant des paires clé-valeur d'en-têtes globaux *pax*.

13.6.2 Les objets *TarInfo*

Un objet `TarInfo` représente un membre dans un `TarFile`. En plus de stocker tous les attributs requis d'un fichier (comme le type de fichier, la taille, l'heure, les autorisations, le propriétaire, etc.), il fournit quelques méthodes utiles pour déterminer son type. Il ne contient pas les données du fichier lui-même.

`TarInfo` objects are returned by `TarFile`'s methods `getmember()`, `getmembers()` and `gettarinfo()`.

Modifying the objects returned by `getmember()` or `getmembers()` will affect all subsequent operations on the archive. For cases where this is unwanted, you can use `copy.copy()` or call the `replace()` method to create a modified copy in one step.

Several attributes can be set to `None` to indicate that a piece of metadata is unused or unknown. Different `TarInfo` methods handle `None` differently :

- The `extract()` or `extractall()` methods will ignore the corresponding metadata, leaving it set to a default.
- `addfile()` will fail.
- `list()` will print a placeholder string.

class `tarfile.TarInfo`(*name=''*)

Crée un objet `TarInfo`.

classmethod `TarInfo.frombuf`(*buf*, *encoding*, *errors*)

Crée et renvoie un objet `TarInfo` à partir de la chaîne tampon *buf*.

Lève `HeaderError` si le tampon n'est pas valide.

classmethod `TarInfo.fromtarfile`(*tarfile*)

Lit le membre suivant dans l'objet `TarFile` *tarfile* et le renvoie comme un objet `TarInfo`.

`TarInfo.tobuf`(*format=DEFAULT_FORMAT*, *encoding=ENCODING*, *errors='surrogateescape'*)

Crée un tampon de chaîne de caractères à partir d'un objet `TarInfo`. Pour plus d'informations sur les arguments, voir le constructeur de la classe `TarFile`.

Modifié dans la version 3.2 : Utilise `'surrogateescape'` comme valeur par défaut pour l'argument *errors*.

Un objet `TarInfo` a les attributs de données publics suivants :

`TarInfo.name: str`

Nom du membre de l'archive.

`TarInfo.size: int`

La taille en octets.

`TarInfo.mtime: int | float`

Time of last modification in seconds since the *epoch*, as in `os.stat_result.st_mtime`.

Modifié dans la version 3.12 : Can be set to `None` for `extract()` and `extractall()`, causing extraction to skip applying this attribute.

`TarInfo.mode: int`

Permission bits, as for `os.chmod()`.

Modifié dans la version 3.12 : Can be set to `None` for `extract()` and `extractall()`, causing extraction to skip applying this attribute.

`TarInfo.type`

Type de fichier. *type* est généralement l'une des constantes suivantes : `REGTYPE`, `AREGTYPE`, `LNKTYPE`, `SYMTYPE`, `DIRTYPE`, `FIFOTYPE`, `CONTTYPE`, `CHRTYPE`, `BLKTYPE`, `GNUTYPE_SPARSE`. Pour déterminer plus facilement le type d'un objet `TarInfo`, utilisez les méthodes `is*()` ci-dessous.

`TarInfo.linkname: str`

Nom du fichier cible, qui n'est présent que dans les objets `TarInfo` de type `LNKTYPE` et `SYMTYPE`.

For symbolic links (`SYMTYPE`), the *linkname* is relative to the directory that contains the link. For hard links (`LNKTYPE`), the *linkname* is relative to the root of the archive.

`TarInfo.uid: int`

ID de l'utilisateur qui a initialement stocké ce membre.

Modifié dans la version 3.12 : Can be set to `None` for `extract()` and `extractall()`, causing extraction to skip applying this attribute.

`TarInfo.gid: int`

ID de groupe de l'utilisateur qui a initialement stocké ce membre.

Modifié dans la version 3.12 : Can be set to `None` for `extract()` and `extractall()`, causing extraction to skip applying this attribute.

`TarInfo.uname: str`

Nom d'utilisateur.

Modifié dans la version 3.12 : Can be set to `None` for `extract()` and `extractall()`, causing extraction to skip applying this attribute.

`TarInfo.gname: str`

Nom de groupe.

Modifié dans la version 3.12 : Can be set to `None` for `extract()` and `extractall()`, causing extraction to skip applying this attribute.

`TarInfo.chksum: int`

Header checksum.

`TarInfo.devmajor: int`

Device major number.

`TarInfo.devminor: int`

Device minor number.

`TarInfo.offset: int`

The tar header starts here.

`TarInfo.offset_data: int`

The file's data starts here.

`TarInfo.sparse`

Sparse member information.

`TarInfo.pax_headers: dict`

Un dictionnaire contenant des paires clé-valeur d'un en-tête étendu *pax* associé.

`TarInfo.replace(name=..., mtime=..., mode=..., linkname=..., uid=..., gid=..., uname=..., gname=..., deep=True)`

Ajouté dans la version 3.12.

Return a *new* copy of the `TarInfo` object with the given attributes changed. For example, to return a `TarInfo` with the group name set to `'staff'`, use :

```
new_tarinfo = old_tarinfo.replace(gname='staff')
```

By default, a deep copy is made. If *deep* is false, the copy is shallow, i.e. `pax_headers` and any custom attributes are shared with the original `TarInfo` object.

Un objet `TarInfo` fournit également des méthodes de requête pratiques :

`TarInfo.isfile()`
Return `True` if the `TarInfo` object is a regular file.

`TarInfo.isreg()`
Identique à `isfile()`.

`TarInfo.isdir()`
Renvoie `True` si c'est un dossier.

`TarInfo.issym()`
Renvoie `True` s'il s'agit d'un lien symbolique.

`TarInfo.islnk()`
Renvoie `True` s'il s'agit d'un lien physique.

`TarInfo.ischr()`
Renvoie `True` s'il s'agit d'un périphérique de caractères.

`TarInfo.isblk()`
Renvoie `True` s'il s'agit d'un périphérique de bloc.

`TarInfo.isfifo()`
Renvoie `True` s'il s'agit d'un tube nommé (*FIFO*).

`TarInfo.isdev()`
Renvoie `True` s'il s'agit d'un périphérique de caractères, d'un périphérique de bloc ou d'un tube nommé.

13.6.3 Extraction filters

Ajouté dans la version 3.12.

The *tar* format is designed to capture all details of a UNIX-like filesystem, which makes it very powerful. Unfortunately, the features make it easy to create tar files that have unintended -- and possibly malicious -- effects when extracted. For example, extracting a tar file can overwrite arbitrary files in various ways (e.g. by using absolute paths, `..` path components, or symlinks that affect later members).

In most cases, the full functionality is not needed. Therefore, *tarfile* supports extraction filters : a mechanism to limit functionality, and thus mitigate some of the security issues.

Avertissement

None of the available filters blocks *all* dangerous archive features. Never extract archives from untrusted sources without prior inspection. See also *Hints for further verification*.

Voir aussi

PEP 706
Contains further motivation and rationale behind the design.

The *filter* argument to `TarFile.extract()` or `extractall()` can be :

— the string `'fully_trusted'` : Honor all metadata as specified in the archive. Should be used if the user trusts the archive completely, or implements their own complex verification.
— the string `'tar'` : Honor most *tar*-specific features (i.e. features of UNIX-like filesystems), but block features that are very likely to be surprising or malicious. See `tar_filter()` for details.
— the string `'data'` : Ignore or block most features specific to UNIX-like filesystems. Intended for extracting cross-platform data archives. See `data_filter()` for details.
— `None` (default) : Use `TarFile.extraction_filter`.

If that is also `None` (the default), the `'data'` filter will be used.

Modifié dans la version 3.14 : The default filter is set to `data`. Previously, the default was equivalent to `fully_trusted`.

— A callable which will be called for each extracted member with a *TarInfo* describing the member and the destination path to where the archive is extracted (i.e. the same path is used for all members) :

```
filter(member: TarInfo, path: str, /) -> TarInfo | None
```

The callable is called just before each member is extracted, so it can take the current state of the disk into account. It can :
— return a `TarInfo` object which will be used instead of the metadata in the archive, or
— return `None`, in which case the member will be skipped, or
— raise an exception to abort the operation or skip the member, depending on `errorlevel`. Note that when extraction is aborted, `extractall()` may leave the archive partially extracted. It does not attempt to clean up.

Default named filters

The pre-defined, named filters are available as functions, so they can be reused in custom filters :

`tarfile.`**`fully_trusted_filter`**(*member*, *path*)

Return *member* unchanged.

This implements the `'fully_trusted'` filter.

`tarfile.`**`tar_filter`**(*member*, *path*)

Implements the `'tar'` filter.
— Strip leading slashes (`/` and `os.sep`) from filenames.
— *Refuse* to extract files with absolute paths (in case the name is absolute even after stripping slashes, e.g. `C:/foo` on Windows). This raises `AbsolutePathError`.
— *Refuse* to extract files whose absolute path (after following symlinks) would end up outside the destination. This raises `OutsideDestinationError`.
— Clear high mode bits (setuid, setgid, sticky) and group/other write bits (`S_IWGRP` | `S_IWOTH`).
Return the modified `TarInfo` member.

`tarfile.`**`data_filter`**(*member*, *path*)

Implements the `'data'` filter. In addition to what `tar_filter` does :
— *Refuse* to extract links (hard or soft) that link to absolute paths, or ones that link outside the destination.

This raises `AbsoluteLinkError` or `LinkOutsideDestinationError`.

Note that such files are refused even on platforms that do not support symbolic links.
— *Refuse* to extract device files (including pipes). This raises `SpecialFileError`.
— For regular files, including hard links :
— Set the owner read and write permissions (`S_IRUSR` | `S_IWUSR`).
— Remove the group & other executable permission (`S_IXGRP` | `S_IXOTH`) if the owner doesn't have it (`S_IXUSR`).
— For other files (directories), set `mode` to `None`, so that extraction methods skip applying permission bits.
— Set user and group info (`uid`, `gid`, `uname`, `gname`) to `None`, so that extraction methods skip setting it.
Return the modified `TarInfo` member.

Note that this filter does not block *all* dangerous archive features. See *Hints for further verification* for details.

Filter errors

When a filter refuses to extract a file, it will raise an appropriate exception, a subclass of `FilterError`. This will abort the extraction if `TarFile.errorlevel` is 1 or more. With `errorlevel=0` the error will be logged and the member will be skipped, but extraction will continue.

Hints for further verification

Even with `filter='data'`, *tarfile* is not suited for extracting untrusted files without prior inspection. Among other issues, the pre-defined filters do not prevent denial-of-service attacks. Users should do additional checks.

Here is an incomplete list of things to consider :

- Extract to a `new temporary directory` to prevent e.g. exploiting pre-existing links, and to make it easier to clean up after a failed extraction.
- When working with untrusted data, use external (e.g. OS-level) limits on disk, memory and CPU usage.
- Check filenames against an allow-list of characters (to filter out control characters, confusables, foreign path separators, and so on).
- Check that filenames have expected extensions (discouraging files that execute when you "click on them", or extension-less files like Windows special device names).
- Limit the number of extracted files, total size of extracted data, filename length (including symlink length), and size of individual files.
- Check for files that would be shadowed on case-insensitive filesystems.

Also note that :

- Tar files may contain multiple versions of the same file. Later ones are expected to overwrite any earlier ones. This feature is crucial to allow updating tape archives, but can be abused maliciously.
- *tarfile* does not protect against issues with "live" data, e.g. an attacker tinkering with the destination (or source) directory while extraction (or archiving) is in progress.

Supporting older Python versions

Extraction filters were added to Python 3.12, but may be backported to older versions as security updates. To check whether the feature is available, use e.g. `hasattr(tarfile, 'data_filter')` rather than checking the Python version.

The following examples show how to support Python versions with and without the feature. Note that setting `extraction_filter` will affect any subsequent operations.

- Fully trusted archive :

```
my_tarfile.extraction_filter = (lambda member, path: member)
my_tarfile.extractall()
```

- Use the `'data'` filter if available, but revert to Python 3.11 behavior (`'fully_trusted'`) if this feature is not available :

```
my_tarfile.extraction_filter = getattr(tarfile, 'data_filter',
                                       (lambda member, path: member))
my_tarfile.extractall()
```

- Use the `'data'` filter ; *fail* if it is not available :

```
my_tarfile.extractall(filter=tarfile.data_filter)
```

ou :

```
my_tarfile.extraction_filter = tarfile.data_filter
my_tarfile.extractall()
```

- Use the `'data'` filter ; *warn* if it is not available :

```
if hasattr(tarfile, 'data_filter'):
    my_tarfile.extractall(filter='data')
else:
```

(suite sur la page suivante)

(suite de la page précédente)

```
    # remove this when no longer needed
    warn_the_user('Extracting may be unsafe; consider updating Python')
    my_tarfile.extractall()
```

Stateful extraction filter example

While *tarfile*'s extraction methods take a simple *filter* callable, custom filters may be more complex objects with an internal state. It may be useful to write these as context managers, to be used like this :

```
with StatefulFilter() as filter_func:
    tar.extractall(path, filter=filter_func)
```

Such a filter can be written as, for example :

```
class StatefulFilter:
    def __init__(self):
        self.file_count = 0

    def __enter__(self):
        return self

    def __call__(self, member, path):
        self.file_count += 1
        return member

    def __exit__(self, *exc_info):
        print(f'{self.file_count} files extracted')
```

13.6.4 Interface en ligne de commande

Ajouté dans la version 3.4.

Le module `tarfile` fournit une interface de ligne de commande simple pour interagir avec les archives *tar*.

Si vous souhaitez créer une nouvelle archive *tar*, spécifiez son nom après l'option `-c`, puis répertorie-le ou les noms de fichiers à inclure :

```
$ python -m tarfile -c monty.tar  spam.txt eggs.txt
```

Passer un répertoire est aussi possible :

```
$ python -m tarfile -c monty.tar life-of-brian_1979/
```

Si vous souhaitez extraire une archive *tar* dans le répertoire courant, utilisez l'option `-e` :

```
$ python -m tarfile -e monty.tar
```

Vous pouvez également extraire une archive *tar* dans un autre répertoire en passant le nom du répertoire :

```
$ python -m tarfile -e monty.tar  other-dir/
```

Pour une liste des fichiers dans une archive *tar*, utilisez l'option `-l` :

```
$ python -m tarfile -l monty.tar
```

Options de la ligne de commande

-l <tarfile>

--list <tarfile>

Liste les fichiers dans une archive *tar*.

-c <tarfile> <source1> ... <sourceN>

--create <tarfile> <source1> ... <sourceN>

Crée une archive *tar* à partir des fichiers sources.

-e <tarfile> [<output_dir>]

--extract <tarfile> [<output_dir>]

Extrait l'archive *tar* dans le répertoire courant si *output_dir* n'est pas spécifié.

-t <tarfile>

--test <tarfile>

Teste si l'archive *tar* est valide ou non.

-v, --verbose

Sortie verbeuse.

--filter <filtername>

Specifies the *filter* for `--extract`. See *Extraction filters* for details. Only string names are accepted (that is, `fully_trusted`, `tar`, and `data`).

13.6.5 Exemples

Comment extraire une archive *tar* dans le dossier de travail courant :

```
import tarfile
tar = tarfile.open("sample.tar.gz")
tar.extractall(filter='data')
tar.close()
```

Comment extraire un sous-ensemble d'une archive *tar* avec `TarFile.extractall()` en utilisant une fonction de générateur au lieu d'une liste :

```
import os
import tarfile

def py_files(members):
    for tarinfo in members:
        if os.path.splitext(tarinfo.name)[1] == ".py":
            yield tarinfo

tar = tarfile.open("sample.tar.gz")
tar.extractall(members=py_files(tar))
tar.close()
```

Comment créer une archive *tar* non compressée à partir d'une liste de noms de fichiers :

```
import tarfile
tar = tarfile.open("sample.tar", "w")
for name in ["foo", "bar", "quux"]:
    tar.add(name)
tar.close()
```

Le même exemple en utilisant l'instruction `with` :

```
import tarfile
with tarfile.open("sample.tar", "w") as tar:
    for name in ["foo", "bar", "quux"]:
        tar.add(name)
```

Comment lire une archive *tar* compressée avec *gzip* et afficher des informations des membres :

```
import tarfile
tar = tarfile.open("sample.tar.gz", "r:gz")
for tarinfo in tar:
    print(tarinfo.name, "is", tarinfo.size, "bytes in size and is ", end="")
    if tarinfo.isreg():
        print("a regular file.")
    elif tarinfo.isdir():
        print("a directory.")
    else:
        print("something else.")
tar.close()
```

Comment créer une archive et réinitialiser les informations de l'utilisateur en utilisant le paramètre *filter* dans `TarFile.add()` :

```
import tarfile
def reset(tarinfo):
    tarinfo.uid = tarinfo.gid = 0
    tarinfo.uname = tarinfo.gname = "root"
    return tarinfo
tar = tarfile.open("sample.tar.gz", "w:gz")
tar.add("foo", filter=reset)
tar.close()
```

13.6.6 Formats *tar* pris en charge

Il existe trois formats *tar* qui peuvent être créés avec le module `tarfile` :

— Le format *POSIX.1-1988 ustar* (`ustar_FORMAT`). Il prend en charge les noms de fichiers jusqu'à une longueur maximale de 256 caractères et les noms de liens jusqu'à 100 caractères. La taille maximale du fichier est de 8 Go. Il s'agit d'un format ancien et limité mais largement pris en charge.

— Le format GNU *tar* (`GNU_FORMAT`). Il prend en charge les noms de fichiers longs et les noms de liens, les fichiers supérieurs à 8 Go et les fichiers discontinus. C'est la norme de facto des systèmes GNU / Linux. `tarfile` prend entièrement en charge les extensions GNU *tar* pour les noms longs, la prise en charge des fichiers discontinus est en lecture seule.

— The POSIX.1-2001 pax format (`PAX_FORMAT`). It is the most flexible format with virtually no limits. It supports long filenames and linknames, large files and stores pathnames in a portable way. Modern tar implementations, including GNU tar, bsdtar/libarchive and star, fully support extended *pax* features ; some old or unmaintained libraries may not, but should treat *pax* archives as if they were in the universally supported *ustar* format. It is the current default format for new archives.

  Il étend le format *ustar* existant avec des en-têtes supplémentaires pour les informations qui ne peuvent pas être stockées autrement. Il existe deux types d'en-têtes *pax* : les en-têtes étendus n'affectent que l'en-tête de fichier suivant, les en-têtes globaux sont valides pour l'archive complète et affectent tous les fichiers suivants. Toutes les données d'un en-tête *pax* sont encodées en *UTF-8* pour des raisons de portabilité.

Il existe d'autres variantes du format *tar* qui peuvent être lues, mais pas créées :

— L'ancien format *V7*. Il s'agit du premier format *tar* d'*Unix Seventh Edition*, ne stockant que des fichiers et répertoires normaux. Les noms ne doivent pas dépasser 100 caractères, il n'y a aucune information de nom d'utilisateur / groupe. Certaines archives ont des sommes de contrôle d'en-tête mal calculées dans le cas de champs avec des caractères non ASCII.

— Format étendu *SunOS tar*. Ce format est une variante du format *POSIX.1-2001 pax*, mais n'est pas compatible.

13.6.7 Problèmes *unicode*

Le format *tar* a été initialement conçu pour effectuer des sauvegardes sur des lecteurs de bande en mettant principalement l'accent sur la préservation des informations du système de fichiers. De nos jours, les archives *tar* sont couramment utilisées pour la distribution de fichiers et l'échange d'archives sur des réseaux. Un problème du format d'origine (qui est la base de tous les autres formats) est qu'il n'existe aucun concept de prise en charge d'encodages de caractères différents. Par exemple, une archive *tar* ordinaire créée sur un système *UTF-8* ne peut pas être lue correctement sur un système *Latin-1* si elle contient des caractères non *ASCII*. Les métadonnées textuelles (comme les noms de fichiers, les noms de liens, les noms d'utilisateurs / de groupes) sembleront endommagées. Malheureusement, il n'y a aucun moyen de détecter automatiquement l'encodage d'une archive. Le format *pax* a été conçu pour résoudre ce problème. Il stocke les métadonnées non ASCII en utilisant l'encodage universel des caractères *UTF-8*.

Les détails de la conversion des caractères dans `tarfile` sont contrôlés par les arguments nommés *encoding* et *errors* de la classe `TarFile`.

encoding définit l'encodage de caractères à utiliser pour les métadonnées de l'archive. La valeur par défaut est `sys.getfilesystemencoding()` ou `'ascii'` comme solution de rechange. Selon que l'archive est lue ou écrite, les métadonnées doivent être décodées ou encodées. Si l'encodage n'est pas défini correctement, cette conversion peut échouer.

L'argument *errors* définit le traitement des caractères qui ne peuvent pas être convertis. Les valeurs possibles sont répertoriées dans la section *Gestionnaires d'erreurs*. Le schéma par défaut est `'surrogateescape'` que Python utilise également pour ses appels de système de fichiers, voir *Noms de fichiers, arguments en ligne de commande, et variables d'environnement*.

Pour les archives `PAX_FORMAT` (par défaut), l'encodage n'est généralement pas nécessaire car toutes les métadonnées sont stockées à l'aide de *UTF-8*. L'encodage n'est utilisé que dans les rares cas où les en-têtes *pax* binaires sont décodés ou lorsque les chaînes avec des caractères de substitution sont stockées.

CHAPITRE 14

Formats de fichiers

Les modules décrits dans ce chapitre lisent divers formats de fichier qui ne sont ni des langages balisés ni relatifs aux e-mails.

14.1 `csv` --- CSV File Reading and Writing

Code source : Lib/csv.py

Le format CSV (*Comma Separated Values*, valeurs séparées par des virgules) est le format le plus commun dans l'importation et l'exportation de feuilles de calculs et de bases de données. Le format fut utilisé pendant des années avant qu'aient lieu des tentatives de standardisation avec la **RFC 4180**. L'absence de format bien défini signifie que des différences subtiles existent dans la production et la consommation de données par différentes applications. Ces différences peuvent gêner lors du traitement de fichiers CSV depuis des sources multiples. Cependant, bien que les séparateurs et délimiteurs varient, le format global est suffisamment similaire pour qu'un module unique puisse manipuler efficacement ces données, masquant au programmeur les détails de lecture/écriture des données.

Le module `csv` implémente des classes pour lire et écrire des données tabulaires au format CSV. Il vous permet de dire « écris ces données dans le format préféré par Excel » ou « lis les données de ce fichier généré par Excel », sans connaître les détails précis du format CSV utilisé par Excel. Vous pouvez aussi décrire les formats CSV utilisés par d'autres applications ou définir vos propres spécialisations.

Les objets `reader` et `writer` du module `csv` lisent et écrivent des séquences. Vous pouvez aussi lire/écrire les données dans un dictionnaire en utilisant les classes `DictReader` et `DictWriter`.

Voir aussi

PEP 305 — Interface des fichiers CSV
La proposition d'amélioration de Python (PEP) qui a proposé cet ajout au langage.

14.1.1 Contenu du module

Le module `csv` définit les fonctions suivantes :

csv.**reader**(*csvfile*, *dialect='excel'*, ***fmtparams*)

Return a *reader object* that will process lines from the given *csvfile*. A csvfile must be an iterable of strings, each in the reader's defined csv format. A csvfile is most commonly a file-like object or list. If *csvfile* is a file object, it should be opened with `newline=''`.[1] An optional *dialect* parameter can be given which is used to define a set of parameters specific to a particular CSV dialect. It may be an instance of a subclass of the `Dialect` class or one of the strings returned by the `list_dialects()` function. The other optional *fmtparams* keyword arguments can be given to override individual formatting parameters in the current dialect. For full details about the dialect and formatting parameters, see section *Dialectes et paramètres de formatage*.

Chaque ligne lue depuis le fichier CSV est renvoyée comme une liste de chaînes de caractères. Aucune conversion automatique de type des données n'est effectuée à moins que l'option de formatage `QUOTE_NONNUMERIC` soit spécifiée (dans ce cas, les champs sans guillemets sont transformés en nombres flottants).

Un court exemple d'utilisation :

```
>>> import csv
>>> with open('eggs.csv', newline='') as csvfile:
...     spamreader = csv.reader(csvfile, delimiter=' ', quotechar='|')
...     for row in spamreader:
...         print(', '.join(row))
Spam, Spam, Spam, Spam, Spam, Baked Beans
Spam, Lovely Spam, Wonderful Spam
```

csv.**writer**(*csvfile*, *dialect='excel'*, ***fmtparams*)

Return a writer object responsible for converting the user's data into delimited strings on the given file-like object. *csvfile* can be any object with a `write()` method. If *csvfile* is a file object, it should be opened with `newline=''`[1]. An optional *dialect* parameter can be given which is used to define a set of parameters specific to a particular CSV dialect. It may be an instance of a subclass of the `Dialect` class or one of the strings returned by the `list_dialects()` function. The other optional *fmtparams* keyword arguments can be given to override individual formatting parameters in the current dialect. For full details about dialects and formatting parameters, see the *Dialectes et paramètres de formatage* section. To make it as easy as possible to interface with modules which implement the DB API, the value `None` is written as the empty string. While this isn't a reversible transformation, it makes it easier to dump SQL NULL data values to CSV files without preprocessing the data returned from a `cursor.fetch*` call. All other non-string data are stringified with `str()` before being written.

Un court exemple d'utilisation :

```
import csv
with open('eggs.csv', 'w', newline='') as csvfile:
    spamwriter = csv.writer(csvfile, delimiter=' ',
                            quotechar='|', quoting=csv.QUOTE_MINIMAL)
    spamwriter.writerow(['Spam'] * 5 + ['Baked Beans'])
    spamwriter.writerow(['Spam', 'Lovely Spam', 'Wonderful Spam'])
```

csv.**register_dialect**(*name*[, *dialect*[, ***fmtparams*]])

Associe *dialect* avec *name*. *name* doit être une chaîne de caractères. Le dialecte peut être spécifié en passant une instance d'une sous-classe de `Dialect`, des arguments nommés *fmtparams*, ou les deux, avec les arguments nommés redéfinissant les paramètres du dialecte. Pour tous les détails sur les dialectes et paramètres de formatage, voir la section *Dialectes et paramètres de formatage*.

csv.**unregister_dialect**(*name*)

Supprime le dialecte associé à *name* depuis le registre des dialectes. Une `Error` est levée si *name* n'est pas un nom de dialecte enregistré.

csv.**get_dialect**(*name*)

Renvoie le dialecte associé à *name*. Une `Error` est levée si *name* n'est pas un nom de dialecte enregistré. Cette

1. Si `newline=''` n'est pas précisé, les caractères de fin de ligne embarqués dans des champs délimités par des guillemets ne seront pas interprétés correctement, et sur les plateformes qui utilisent `\r\n` comme marqueur de fin de ligne, un `\r` sera ajouté. Il devrait toujours être sûr de préciser `newline=''`, puisque le module *csv* gère lui-même les fins de lignes (*universelles*).

fonction renvoie un objet `Dialect` immuable.

`csv.list_dialects()`

Renvoie les noms de tous les dialectes enregistrés.

`csv.field_size_limit([new_limit])`

Renvoie la taille de champ maximale actuelle autorisée par l'analyseur. Si *new_limit* est donnée, elle devient la nouvelle limite.

Le module `csv` définit les classes suivantes :

class `csv.DictReader(f, fieldnames=None, restkey=None, restval=None, dialect='excel', *args, **kwds)`

Crée un objet qui opère comme un lecteur ordinaire mais assemble les informations de chaque ligne dans un `dict` dont les clés sont données par le paramètre optionnel *fieldnames*.

The *fieldnames* parameter is a *sequence*. If *fieldnames* is omitted, the values in the first row of file *f* will be used as the fieldnames and will be omitted from the results. If *fieldnames* is provided, they will be used and the first row will be included in the results. Regardless of how the fieldnames are determined, the dictionary preserves their original ordering.

Si une ligne a plus de champs que *fieldnames*, les données excédentaires sont mises dans une liste stockée dans le champ spécifié par *restkey* (`None` par défaut). Si une ligne non-vide a moins de champs que *fieldnames*, les valeurs manquantes sont remplacées par la valeur de *restval* (`None` par défaut).

Tous les autres arguments optionnels ou nommés sont passés à l'instance `reader` sous-jacente.

If the argument passed to *fieldnames* is an iterator, it will be coerced to a `list`.

Modifié dans la version 3.6 : Les lignes renvoyées sont maintenant de type `OrderedDict`.

Modifié dans la version 3.8 : Les lignes renvoyées sont maintenant de type `dict`.

Un court exemple d'utilisation :

```
>>> import csv
>>> with open('names.csv', newline='') as csvfile:
...     reader = csv.DictReader(csvfile)
...     for row in reader:
...         print(row['first_name'], row['last_name'])
...
Eric Idle
John Cleese

>>> print(row)
{'first_name': 'John', 'last_name': 'Cleese'}
```

class `csv.DictWriter(f, fieldnames, restval='', extrasaction='raise', dialect='excel', *args, **kwds)`

Create an object which operates like a regular writer but maps dictionaries onto output rows. The *fieldnames* parameter is a `sequence` of keys that identify the order in which values in the dictionary passed to the `writerow()` method are written to file *f*. The optional *restval* parameter specifies the value to be written if the dictionary is missing a key in *fieldnames*. If the dictionary passed to the `writerow()` method contains a key not found in *fieldnames*, the optional *extrasaction* parameter indicates what action to take. If it is set to `'raise'`, the default value, a `ValueError` is raised. If it is set to `'ignore'`, extra values in the dictionary are ignored. Any other optional or keyword arguments are passed to the underlying `writer` instance.

Notez que contrairement à la classe `DictReader`, le paramètre *fieldnames* de `DictWriter` n'est pas optionnel.

If the argument passed to *fieldnames* is an iterator, it will be coerced to a `list`.

Un court exemple d'utilisation :

```
import csv
```

(suite sur la page suivante)

(suite de la page précédente)

```
with open('names.csv', 'w', newline='') as csvfile:
    fieldnames = ['first_name', 'last_name']
    writer = csv.DictWriter(csvfile, fieldnames=fieldnames)

    writer.writeheader()
    writer.writerow({'first_name': 'Baked', 'last_name': 'Beans'})
    writer.writerow({'first_name': 'Lovely', 'last_name': 'Spam'})
    writer.writerow({'first_name': 'Wonderful', 'last_name': 'Spam'})
```

class csv.**Dialect**

La classe *Dialect* est une classe dont les attributs contiennent des informations sur la façon de gérer les guillemets, les espaces, les délimiteurs, etc. En raison de l'absence d'une spécification CSV stricte, différentes applications produisent des données CSV subtilement différentes. Les instances *Dialect* définissent le comportement des instances *reader* et *writer*.

Tous les noms disponibles de *Dialect* sont renvoyés par *list_dialects()*, et ils peuvent être enregistrés avec des classes *reader* et *writer* spécifiques en passant par leur fonction d'initialisation (`__init__`) comme ici :

```
import csv

with open('students.csv', 'w', newline='') as csvfile:
    writer = csv.writer(csvfile, dialect='unix')
```

class csv.**excel**

La classe *excel* définit les propriétés usuelles d'un fichier CSV généré par Excel. Elle est enregistrée avec le nom de dialecte `'excel'`.

class csv.**excel_tab**

La classe *excel_tab* définit les propriétés usuelles d'un fichier CSV généré par Excel avec des tabulations comme séparateurs. Elle est enregistrée avec le nom de dialecte `'excel-tab'`.

class csv.**unix_dialect**

La classe *unix_dialect* définit les propriétés usuelles d'un fichier CSV généré sur un système Unix, c'est-à-dire utilisant `'\n'` comme marqueur de fin de ligne et délimitant tous les champs par des guillemets. Elle est enregistrée avec le nom de dialecte `'unix'`.

Ajouté dans la version 3.2.

class csv.**Sniffer**

La classe *Sniffer* est utilisée pour déduire le format d'un fichier CSV.

La classe *Sniffer* fournit deux méthodes :

sniff(*sample*, *delimiters=None*)

Analyse l'extrait donné (*sample*) et renvoie une sous-classe *Dialect* reflétant les paramètres trouvés. Si le paramètre optionnel *delimiters* est donné, il est interprété comme une chaîne contenant tous les caractères valides de séparation possibles.

has_header(*sample*)

Analyse l'extrait de texte (présumé être au format CSV) et renvoie *True* si la première ligne semble être une série d'en-têtes de colonnes. En inspectant chaque colonne, l'un des deux critères clés sera pris en compte pour estimer si l'échantillon contient un en-tête :

— les deuxième à n-ième lignes contiennent des valeurs numériques
— les deuxième à n-ième lignes contiennent des chaînes dont la longueur d'au moins une valeur diffère de celle de l'en-tête putatif de cette colonne.

L'échantillon est composé des vingt lignes après la première ligne ; si plus de la moitié des colonnes + lignes répondent aux critères, *True* est renvoyé.

> **Note**
>
> Cette méthode est une heuristique discutable et peut produire tant des faux-positifs que des faux-négatifs.

Un exemple d'utilisation de `Sniffer` :

```
with open('example.csv', newline='') as csvfile:
    dialect = csv.Sniffer().sniff(csvfile.read(1024))
    csvfile.seek(0)
    reader = csv.reader(csvfile, dialect)
    # ... process CSV file contents here ...
```

Le module `csv` définit les constantes suivantes :

`csv.`**`QUOTE_ALL`**

Indique aux objets `writer` de délimiter tous les champs par des guillemets.

`csv.`**`QUOTE_MINIMAL`**

Indique aux objets `writer` de ne délimiter ainsi que les champs contenant un caractère spécial comme *delimiter*, *quotechar* ou n'importe quel caractère de *lineterminator*.

`csv.`**`QUOTE_NONNUMERIC`**

Indique aux objets `writer` de délimiter ainsi tous les champs non-numériques.

Instructs `reader` objects to convert all non-quoted fields to type *float*.

`csv.`**`QUOTE_NONE`**

Indique aux objets `writer` de ne jamais délimiter les champs par des guillemets. Quand le *delimiter* utilisé apparaît dans les données, il est précédé sur la sortie par un caractère *escapechar*. Si *escapechar* n'est pas précisé, le transcripteur lèvera une `Error` si un caractère nécessitant un échappement est rencontré.

Instructs `reader` objects to perform no special processing of quote characters.

`csv.`**`QUOTE_NOTNULL`**

Instructs `writer` objects to quote all fields which are not `None`. This is similar to `QUOTE_ALL`, except that if a field value is `None` an empty (unquoted) string is written.

Instructs `reader` objects to interpret an empty (unquoted) field as `None` and to otherwise behave as `QUOTE_ALL`.

Ajouté dans la version 3.12.

`csv.`**`QUOTE_STRINGS`**

Instructs `writer` objects to always place quotes around fields which are strings. This is similar to `QUOTE_NONNUMERIC`, except that if a field value is `None` an empty (unquoted) string is written.

Instructs `reader` objects to interpret an empty (unquoted) string as `None` and to otherwise behave as `QUOTE_NONNUMERIC`.

Ajouté dans la version 3.12.

Le module `csv` définit les exceptions suivantes :

`exception` `csv.`**`Error`**

Levée par les fonctions du module quand une erreur détectée.

14.1.2 Dialectes et paramètres de formatage

To make it easier to specify the format of input and output records, specific formatting parameters are grouped together into dialects. A dialect is a subclass of the `Dialect` class containing various attributes describing the format of the CSV file. When creating `reader` or `writer` objects, the programmer can specify a string or a subclass of the `Dialect` class as the dialect parameter. In addition to, or instead of, the *dialect* parameter, the programmer can

also specify individual formatting parameters, which have the same names as the attributes defined below for the `Dialect` class.

Les dialectes supportent les attributs suivants :

`Dialect.`**`delimiter`**

Une chaîne d'un seul caractère utilisée pour séparer les champs. Elle vaut `','` par défaut.

`Dialect.`**`doublequote`**

Contrôle comment les caractères *quotechar* dans le champ doivent être retranscrits. Quand ce paramètre vaut `True`, le caractère est doublé. Quand il vaut `False`, le caractère *escapechar* est utilisé comme préfixe à *quotechar*. Il vaut `True` par défaut.

En écriture, si *doublequote* vaut `False` et qu'aucun *escapechar* n'est précisé, une `Error` est levée si un *quotechar* est trouvé dans le champ.

`Dialect.`**`escapechar`**

Une chaîne d'un seul caractère utilisée par le transcripteur pour échapper *delimiter* si *quoting* vaut `QUOTE_NONE`, et pour échapper *quotechar* si *doublequote* vaut `False`. À la lecture, *escapechar* retire toute signification spéciale au caractère qui le suit. Elle vaut par défaut `None`, ce qui désactive l'échappement.

Modifié dans la version 3.11 : Un `escapechar` vide n'est pas autorisé.

`Dialect.`**`lineterminator`**

La chaîne utilisée pour terminer les lignes produites par un `writer`. Elle vaut par défaut `'\r\n'`.

> **Note**
>
> La classe `reader` est codée en dur pour reconnaître `'\r'` et `'\n'` comme marqueurs de fin de ligne, et ignorer *lineterminator*. Ce comportement pourrait changer dans le futur.

`Dialect.`**`quotechar`**

Une chaîne d'un seul caractère utilisée pour délimiter les champs contenant des caractères spéciaux, comme *delimiter* ou *quotechar*, ou contenant un caractère de fin de ligne. Elle vaut `'"'` par défaut.

Modifié dans la version 3.11 : Un `quotechar` vide n'est pas autorisé.

`Dialect.`**`quoting`**

Controls when quotes should be generated by the writer and recognised by the reader. It can take on any of the *QUOTE_* constants* and defaults to `QUOTE_MINIMAL`.

`Dialect.`**`skipinitialspace`**

Quand il vaut `True`, les espaces suivant directement *delimiter* sont ignorés. Il vaut `False` par défaut.

`Dialect.`**`strict`**

Quand il vaut `True`, une exception `Error` est levée lors de mauvaises entrées CSV. Il vaut `False` par défaut.

14.1.3 Objets lecteurs

Les objets lecteurs (instances de `DictReader` ou objets renvoyés par la fonction `reader()`) ont les méthodes publiques suivantes :

`csvreader.`**`__next__`**`()`

Renvoie la ligne suivante de l'objet itérable du lecteur en tant que liste (si l'objet est renvoyé depuis `reader()`) ou dictionnaire (si l'objet est un `DictReader`), analysé suivant la classe `Dialect` utilisée. Généralement, vous devez appeler la méthode à l'aide de `next(reader)`.

Les objets lecteurs ont les attributs publics suivants :

`csvreader.`**`dialect`**

Une description en lecture seule du dialecte utilisé par l'analyseur.

csvreader.**line_num**

Le nombre de lignes lues depuis l'itérateur source. Ce n'est pas équivalent au nombre d'enregistrements renvoyés, puisque certains enregistrements peuvent s'étendre sur plusieurs lignes.

Les objets *DictReader* ont les attributs publics suivants :

DictReader.**fieldnames**

S'il n'est pas passé comme paramètre à la création de l'objet, cet attribut est initialisé lors du premier accès ou quand le premier enregistrement est lu depuis le fichier.

14.1.4 Objets transcripteurs

`writer` objects (`DictWriter` instances and objects returned by the `writer()` function) have the following public methods. A *row* must be an iterable of strings or numbers for `writer` objects and a dictionary mapping fieldnames to strings or numbers (by passing them through `str()` first) for `DictWriter` objects. Note that complex numbers are written out surrounded by parens. This may cause some problems for other programs which read CSV files (assuming they support complex numbers at all).

csvwriter.**writerow**(*row*)

Écrit le paramètre *row* vers l'objet fichier du transcripteur, formaté selon la classe `Dialect` utilisée. Renvoie la valeur de retour de l'appel à la méthode *write* de l'objet fichier sous-jacent.

Modifié dans la version 3.5 : Ajout du support d'itérables arbitraires.

csvwriter.**writerows**(*rows*)

Écrit tous les éléments de *rows* (itérable d'objets *row* comme décrits précédemment) vers le fichier associé au transcripteur, formatés selon le dialecte utilisé.

Les objets transcripteurs ont l'attribut public suivant :

csvwriter.**dialect**

Une description en lecture seule du dialecte utilisé par le transcripteur.

Les objets *DictWriter* ont la méthode publique suivante :

DictWriter.**writeheader**()

Écrit une ligne avec le nom des en-têtes (comme définies dans le constructeur) dans l'objet fichier associé au transcripteur, formatée selon le dialecte utilisé. Renvoie la valeur de retour de l'appel `csvwriter.writerow()` utilisé en interne.

Ajouté dans la version 3.2.

Modifié dans la version 3.8 : `writeheader()` renvoie maintenant aussi la valeur renvoyée par la méthode `csvwriter.writerow()` qu'il utilise en interne.

14.1.5 Exemples

Le plus simple exemple de lecture d'un fichier CSV :

```
import csv
with open('some.csv', newline='') as f:
    reader = csv.reader(f)
    for row in reader:
        print(row)
```

Lire un fichier avec un format alternatif :

```
import csv
with open('passwd', newline='') as f:
    reader = csv.reader(f, delimiter=':', quoting=csv.QUOTE_NONE)
    for row in reader:
        print(row)
```

Le plus simple exemple d'écriture correspondant est :

```
import csv
with open('some.csv', 'w', newline='') as f:
    writer = csv.writer(f)
    writer.writerows(someiterable)
```

Puisque `open()` est utilisée pour ouvrir un fichier CSV en lecture, le fichier sera par défaut décodé vers Unicode en utilisant l'encodage par défaut (voir `locale.getencoding()`). Pour décoder un fichier utilisant un encodage différent, utilisez l'argument `encoding` de *open* :

```
import csv
with open('some.csv', newline='', encoding='utf-8') as f:
    reader = csv.reader(f)
    for row in reader:
        print(row)
```

Cela s'applique également lors de l'écriture dans un autre encodage que celui par défaut du système : spécifiez l'encodage en argument lors de l'ouverture du fichier de sortie.

Enregistrer un nouveau dialecte :

```
import csv
csv.register_dialect('unixpwd', delimiter=':', quoting=csv.QUOTE_NONE)
with open('passwd', newline='') as f:
    reader = csv.reader(f, 'unixpwd')
```

Un exemple d'utilisation un peu plus avancé du lecteur --- attrapant et notifiant les erreurs :

```
import csv, sys
filename = 'some.csv'
with open(filename, newline='') as f:
    reader = csv.reader(f)
    try:
        for row in reader:
            print(row)
    except csv.Error as e:
        sys.exit('file {}, line {}: {}'.format(filename, reader.line_num, e))
```

Et bien que le module ne permette pas d'analyser directement des chaînes, cela peut être fait facilement :

```
import csv
for row in csv.reader(['one,two,three']):
    print(row)
```

Notes

14.2 `configparser` --- Configuration file parser

Code source : Lib/configparser.py

Ce module fournit la classe `ConfigParser`. Cette classe implémente un langage de configuration basique, proche de ce que l'on peut trouver dans les fichiers *INI* de Microsoft Windows. Vous pouvez utiliser ce module pour écrire des programmes Python qui sont facilement configurables par l'utilisateur final.

Note

Ce module *n'implémente pas* la version étendue de la syntaxe *INI* qui permet de lire ou d'écrire des valeurs dans la base de registre Windows en utilisant divers préfixes.

Voir aussi

Module `tomlib`
TOML est un format bien spécifié pour les fichiers de configuration d'application. Il est spécifiquement conçu pour être une version améliorée d'INI.

Module *`shlex`*
Support for creating Unix shell-like mini-languages which can also be used for application configuration files.

Module *`json`*
The `json` module implements a subset of JavaScript syntax which is sometimes used for configuration, but does not support comments.

14.2.1 Premiers pas

Prenons pour exemple un fichier de configuration très simple ressemblant à ceci :

```
[DEFAULT]
ServerAliveInterval = 45
Compression = yes
CompressionLevel = 9
ForwardX11 = yes

[forge.example]
User = hg

[topsecret.server.example]
Port = 50022
ForwardX11 = no
```

La structure des fichiers *INI* est décrite dans la *section suivante*. En bref, chaque fichier est constitué de sections, chacune des sections comprenant des clés associées à des valeurs. Les classes du module *`configparser`* peuvent écrire et lire de tels fichiers. Commençons par le code qui permet de générer le fichier ci-dessus.

```
>>> import configparser
>>> config = configparser.ConfigParser()
>>> config['DEFAULT'] = {'ServerAliveInterval': '45',
...                      'Compression': 'yes',
...                      'CompressionLevel': '9'}
>>> config['forge.example'] = {}
>>> config['forge.example']['User'] = 'hg'
>>> config['topsecret.server.example'] = {}
>>> topsecret = config['topsecret.server.example']
>>> topsecret['Port'] = '50022'     # mutates the parser
>>> topsecret['ForwardX11'] = 'no'  # same here
>>> config['DEFAULT']['ForwardX11'] = 'yes'
>>> with open('example.ini', 'w') as configfile:
...   config.write(configfile)
...
```

Comme vous pouvez le voir, nous pouvons manipuler l'instance renvoyée par l'analyse du fichier de configuration

comme s'il s'agissait d'un dictionnaire. Il y a des différences, comme *explicité ci-dessous*, mais le comportement de l'instance est très proche de ce que vous pourriez attendre d'un dictionnaire.

Nous venons de créer et sauvegarder un fichier de configuration. Voyons maintenant comment nous pouvons le lire et accéder aux données qu'il contient.

```
>>> config = configparser.ConfigParser()
>>> config.sections()
[]
>>> config.read('example.ini')
['example.ini']
>>> config.sections()
['forge.example', 'topsecret.server.example']
>>> 'forge.example' in config
True
>>> 'python.org' in config
False
>>> config['forge.example']['User']
'hg'
>>> config['DEFAULT']['Compression']
'yes'
>>> topsecret = config['topsecret.server.example']
>>> topsecret['ForwardX11']
'no'
>>> topsecret['Port']
'50022'
>>> for key in config['forge.example']:
...     print(key)
user
compressionlevel
serveraliveinterval
compression
forwardx11
>>> config['forge.example']['ForwardX11']
'yes'
```

Comme vous le voyez, l'API est assez simple à utiliser. La seule partie un peu magique concerne la section `DEFAULT`, qui fournit les valeurs par défaut pour toutes les autres sections [1]. Notez également que les clés à l'intérieur des sections ne sont pas sensibles à la casse et qu'elles sont stockées en minuscules.[1].

Il est possible de lire plusieurs configurations dans un seul objet *`ConfigParser`*, où la configuration ajoutée la plus récemment a la priorité la plus élevée. Toutes les clés en conflit sont prises de la configuration la plus récente tandis que les anciennes clés déjà existantes sont mises de côté. L'exemple suivant lit un fichier `override.ini` qui prendra le dessus sur toutes les clés en conflit avec le fichier `exemple.ini`.

```
[DEFAULT]
ServerAliveInterval = -1
```

```
>>> config_override = configparser.ConfigParser()
>>> config_override['DEFAULT'] = {'ServerAliveInterval': '-1'}
>>> with open('override.ini', 'w') as configfile:
...     config_override.write(configfile)
...
>>> config_override = configparser.ConfigParser()
>>> config_override.read(['example.ini', 'override.ini'])
['example.ini', 'override.ini']
```

(suite sur la page suivante)

1. Config parsers allow for heavy customization. If you are interested in changing the behaviour outlined by the footnote reference, consult the *Customizing Parser Behaviour* section.

(suite de la page précédente)

```
>>> print(config_override.get('DEFAULT', 'ServerAliveInterval'))
-1
```

Ce comportement est équivalent à un appel à `ConfigParser.read()` avec plusieurs fichiers passés dans le paramètre *filenames*.

14.2.2 Types de données prises en charge

Les lecteurs de configuration n'essayent jamais de deviner le type des valeurs présentes dans les fichiers de configuration, et elles sont toujours stockées en tant que chaînes de caractères. Ainsi, si vous avez besoin d'un type différent, vous devez effectuer la conversion vous-même :

```
>>> int(topsecret['Port'])
50022
>>> float(topsecret['CompressionLevel'])
9.0
```

Puisque que cette tâche doit être fréquemment accomplie, les lecteurs de configurations fournissent un ensemble d'accesseurs permettant de gérer les entiers, les flottants et les booléens plus facilement. Le cas des booléens est le plus pertinent. En effet, vous ne pouvez pas vous contenter d'utiliser la fonction `bool()` directement puisque `bool('False')` renvoie `True`. C'est pourquoi les lecteurs fournissent également la méthode `getboolean()`. Cette méthode n'est pas sensible à la casse et interprète correctement les valeurs booléennes associées aux chaînes de caractères comme `'yes'`-`'no'`, `'on'`-`'off'`, `'true'`-`'false'` et `'1'`-`'0'`[page 622, 1]. Par exemple :

```
>>> topsecret.getboolean('ForwardX11')
False
>>> config['forge.example'].getboolean('ForwardX11')
True
>>> config.getboolean('forge.example', 'Compression')
True
```

En plus de `getboolean()`, les lecteurs de configurations fournissent également des méthodes similaires comme `getint()` et `getfloat()`. Vous pouvez enregistrer vos propres convertisseurs et personnaliser ceux déjà fournis.[page 622, 1]

14.2.3 Valeurs de substitution

As with a dictionary, you can use a section's `get()` method to provide fallback values :

```
>>> topsecret.get('Port')
'50022'
>>> topsecret.get('CompressionLevel')
'9'
>>> topsecret.get('Cipher')
>>> topsecret.get('Cipher', '3des-cbc')
'3des-cbc'
```

Please note that default values have precedence over fallback values. For instance, in our example the `'CompressionLevel'` key was specified only in the `'DEFAULT'` section. If we try to get it from the section `'topsecret.server.example'`, we will always get the default, even if we specify a fallback :

```
>>> topsecret.get('CompressionLevel', '3')
'9'
```

One more thing to be aware of is that the parser-level `get()` method provides a custom, more complex interface, maintained for backwards compatibility. When using this method, a fallback value can be provided via the `fallback` keyword-only argument :

```
>>> config.get('forge.example', 'monster',
...            fallback='No such things as monsters')
'No such things as monsters'
```

L'argument `fallback` peut être utilisé de la même façon avec les méthodes *getint()*, *getfloat()* et *getboolean()*. Par exemple :

```
>>> 'BatchMode' in topsecret
False
>>> topsecret.getboolean('BatchMode', fallback=True)
True
>>> config['DEFAULT']['BatchMode'] = 'no'
>>> topsecret.getboolean('BatchMode', fallback=True)
False
```

14.2.4 Structure des fichiers *INI* prise en change

A configuration file consists of sections, each led by a `[section]` header, followed by key/value entries separated by a specific string (= or : by default[page 622, 1]). By default, section names are case sensitive but keys are not[page 622, 1]. Leading and trailing whitespace is removed from keys and values. Values can be omitted if the parser is configured to allow it[page 622, 1], in which case the key/value delimiter may also be left out. Values can also span multiple lines, as long as they are indented deeper than the first line of the value. Depending on the parser's mode, blank lines may be treated as parts of multiline values or ignored.

Par défaut, un nom de section valide peut-être n'importe quelle chaîne de caractères ne contenant pas le caractère `'\\n'`. Pour modifier cela, voir *ConfigParser.SECTCRE*.

The first section name may be omitted if the parser is configured to allow an unnamed top level section with `allow_unnamed_section=True`. In this case, the keys/values may be retrieved by *UNNAMED_SECTION* as in `config[UNNAMED_SECTION]`.

Les fichiers de configuration peuvent contenir des commentaires, préfixés par des caractères spécifiques (`#` et `;` par défaut[page 622, 1]). Les commentaires peuvent apparaître à l'emplacement d'une ligne vide, et peuvent aussi être indentés.[page 622, 1]

Par exemple :

```
[Simple Values]
key=value
spaces in keys=allowed
spaces in values=allowed as well
spaces around the delimiter = obviously
you can also use : to delimit keys from values

[All Values Are Strings]
values like this: 1000000
or this: 3.14159265359
are they treated as numbers? : no
integers, floats and booleans are held as: strings
can use the API to get converted values directly: true

[Multiline Values]
chorus: I'm a lumberjack, and I'm okay
    I sleep all night and I work all day

[No Values]
key_without_value
empty string value here =
```

(suite sur la page suivante)

(suite de la page précédente)

```
[You can use comments]
# like this
; or this

# By default only in an empty line.
# Inline comments can be harmful because they prevent users
# from using the delimiting characters as parts of values.
# That being said, this can be customized.

    [Sections Can Be Indented]
        can_values_be_as_well = True
        does_that_mean_anything_special = False
        purpose = formatting for readability
        multiline_values = are
            handled just fine as
            long as they are indented
            deeper than the first line
            of a value
        # Did I mention we can indent comments, too?
```

14.2.5 Unnamed Sections

The name of the first section (or unique) may be omitted and values retrieved by the *UNNAMED_SECTION* attribute.

```
>>> config = """
... option = value
...
... [  Section 2  ]
... another = val
... """
>>> unnamed = configparser.ConfigParser(allow_unnamed_section=True)
>>> unnamed.read_string(config)
>>> unnamed.get(configparser.UNNAMED_SECTION, 'option')
'value'
```

14.2.6 Interpolation des valeurs

La classe *ConfigParser* prend en charge l'interpolation, en plus des fonctionnalités de base. Cela signifie que les valeurs peuvent être traitées avant d'être renvoyées par les appels aux méthodes `get()`.

class configparser.**BasicInterpolation**

Implémentation par défaut utilisée par la classe *ConfigParser*. Celle-ci permet aux valeurs de contenir des chaînes de formatage se référant à d'autres valeurs dans la même section, ou bien à des valeurs dans la section spéciale par défaut[page 622, 1]. D'autres valeurs par défaut peuvent être fournies au moment de l'initialisation de cette classe.

Par exemple :

```
[Paths]
home_dir: /Users
my_dir: %(home_dir)s/lumberjack
my_pictures: %(my_dir)s/Pictures

[Escape]
# use a %% to escape the % sign (% is the only character that needs to be␣
↪escaped):
gain: 80%%
```

Dans l'exemple ci-dessus, une classe `Configparser` dont l'attribut *interpolation* vaut `BasicInterpolation()` interprète la chaîne de caractères `%(home_dir)s` en utilisant la valeur de la clé `home_dir` (`/Users` dans ce cas). `%(my_dir)s` est interprétée comme `/Users/lumberjack`. Les interpolations sont effectuées à la volée. Ainsi, les clés utilisées comme référence à l'intérieur des chaînes de formatage peuvent être définies dans le fichier de configuration dans n'importe quel ordre.

Si l'attribut `interpolation` vaut `None`, le lecteur renvoie `%(my_dir)s/Pictures` comme valeur pour `my_pictures` et `%(home_dir)s/lumberjack` comme valeur pour `my_dir`.

class `configparser.`**`ExtendedInterpolation`**

Autre façon de gérer l'interpolation en utilisant une syntaxe plus avancée, utilisée par exemple par `zc.buildout`. Cette syntaxe étendue utilise la chaîne de formatage `{section:option}}` pour désigner une valeur appartenant à une autre section. L'interpolation peut s'étendre sur plusieurs niveaux. Par commodité, si la partie `{section}` est absente, l'interpolation utilise la section courante par défaut (et, le cas échéant, les valeurs de la section par défaut spéciale).

Voici comment transformer la configuration ci-dessus avec la syntaxe d'interpolation étendue :

```
[Paths]
home_dir: /Users
my_dir: ${home_dir}/lumberjack
my_pictures: ${my_dir}/Pictures

[Escape]
# use a $$ to escape the $ sign ($ is the only character that needs to be␣
↪escaped):
cost: $$80
```

Vous pouvez également récupérer des valeurs appartenant aux autres sections :

```
[Common]
home_dir: /Users
library_dir: /Library
system_dir: /System
macports_dir: /opt/local

[Frameworks]
Python: 3.2
path: ${Common:system_dir}/Library/Frameworks/

[Arthur]
nickname: Two Sheds
last_name: Jackson
my_dir: ${Common:home_dir}/twosheds
my_pictures: ${my_dir}/Pictures
python_dir: ${Frameworks:path}/Python/Versions/${Frameworks:Python}
```

14.2.7 Protocole d'accès associatif

Ajouté dans la version 3.2.

Le terme « protocole d'accès associatif » est utilisé pour décrire la fonctionnalité qui permet d'utiliser des objets personnalisés comme s'il s'agissait de dictionnaires. Dans le cas du module *`configparser`*, l'implémentation du protocole utilise la notation `parser['section']['option']`.

En particulier, `parser['section']` renvoie un mandataire vers les données de la section correspondantes dans l'analyseur. Cela signifie que les valeurs ne sont pas copiées, mais prélevées depuis l'analyseur initial à la demande. Plus important encore, lorsque les valeurs sont changées dans un mandataire pour une section, elles sont en réalité changées dans l'analyseur initial.

Les objets du module *`configparser`* se comportent le plus possible comme des vrais dictionnaires. L'interface

est complète et suit les définitions fournies par la classe abstraite `MutableMapping`. Cependant, il faut prendre en compte un certain nombre de différences :

— Par défaut, toutes les clés des sections sont accessibles sans respect de la casse[page 622, 1]. Par exemple, `for option in parser["section"]` renvoie uniquement les clés telles que transformées par la méthode `optionxform`, c'est-à-dire des clés transformées en minuscules. De même, pour une section contenant la clé `a`, les deux expressions suivantes renvoient `True` :

```
"a" in parser["section"]
"A" in parser["section"]
```

— Toutes les sections incluent en plus les valeurs de la section `DEFAULTSECT`. Cela signifie qu'appeler `clear()` sur une section ne la fera pas forcément apparaître vide. En effet, les valeurs par défaut ne peuvent pas être supprimées de la section (car, techniquement, elles n'y sont pas présentes). Si vous détruisez une valeur par défaut qui a été écrasée dans une section, alors la valeur par défaut sera de nouveau visible. Essayer de détruire une valeur par défaut lève l'exception `KeyError`.

— La section `DEFAULTSECT` ne peut pas être supprimée :
  — l'exception `ValueError` est levée si on essaye de la supprimer ;
  — appeler `parser.clear()` la laisse intacte ;
  — appeler `parser.popitem()` ne la renvoie jamais.

— Le deuxième argument de `parser.get(section, option, **kwargs)` n'est **pas** une valeur de substitution. Notez cependant que les méthodes `get()` fournies par les sections sont compatibles à la fois avec le protocole associatif et avec l'API classique de *configparser*.

— La méthode `parser.items()` est compatible avec le protocole d'accès associatif et renvoie une liste de paires *section_name*, *section_proxy*, en incluant la section *DEFAULTSECT*. Cependant, cette méthode peut aussi être appelée avec des arguments : `parser.items(section, raw, vars)`. Dans ce cas, la méthode renvoie une liste de paires *option*, *value* pour la section spécifiée, en interprétant les interpolations (à moins d'utiliser `raw=True`).

Le protocole d'accès est implémenté au-dessus de l'ancienne API. Ainsi, les sous-classes qui écrasent des méthodes de l'interface originale se comportent correctement du point de vue du protocole d'accès.

14.2.8 Personnalisation du comportement de l'analyseur

Il existe pratiquement autant de variations du format *INI* que d'applications qui l'utilisent. Le module `configparser` fait son possible pour gérer le plus grand nombre de variantes raisonnables du style *INI*. Le comportement par défaut est principalement contraint par des raisons historiques. De ce fait, il est très probable qu'il soit nécessaire de personnaliser certaines des fonctionnalités de ce module.

The most common way to change the way a specific config parser works is to use the `__init__()` options :

— *defaults*, valeur par défaut : `None`

  Cette option accepte un dictionnaire de paires clé—valeurs qui seront placées dans la section `DEFAULT` initialement. Ceci est une façon élégante de prendre en charge des fichiers de configuration qui n'ont pas besoin de spécifier de valeurs lorsque celles-ci sont identiques aux valeurs par défaut documentées.

  Hint : if you want to specify default values for a specific section, use `read_dict()` before you read the actual file.

— *dict_type*, valeur par défaut : `dict`

  Cette option influe de manière importante sur la façon dont le protocole d'accès associatif se comporte et ce à quoi ressemblent les fichiers de configuration une fois écrits. Avec un dictionnaire standard, les sections sont stockées dans l'ordre où elles ont été ajoutées à l'analyseur. Ceci est également vrai pour les options à l'intérieur des sections.

  Si vous souhaitez classer les sections et les options lors de l'écriture par exemple, vous pouvez utiliser un type de dictionnaire différent.

  À noter : il est possible d'ajouter un ensemble de paires clés—valeurs en une seule opération. L'ordre des clés est préservé si vous utilisez un dictionnaire standard pour cela. Par exemple :

```
>>> parser = configparser.ConfigParser()
>>> parser.read_dict({'section1': {'key1': 'value1',
```

(suite sur la page suivante)

(suite de la page précédente)

```
...                                 'key2': 'value2',
...                                 'key3': 'value3'},
...                   'section2': {'keyA': 'valueA',
...                                 'keyB': 'valueB',
...                                 'keyC': 'valueC'},
...                   'section3': {'foo': 'x',
...                                 'bar': 'y',
...                                 'baz': 'z'}
... })
>>> parser.sections()
['section1', 'section2', 'section3']
>>> [option for option in parser['section3']]
['foo', 'bar', 'baz']
```

— *allow_no_value*, valeur par défaut : `False`

Certains fichiers de configurations sont connus pour contenir des options sans valeur associée, tout en se conformant à la syntaxe prise en charge par le module `configparser` par ailleurs. Pour indiquer que de telles valeurs sont acceptables, utilisez le paramètre *allow_no_value* lors de la construction de l'instance :

```
>>> import configparser

>>> sample_config = """
... [mysqld]
...   user = mysql
...   pid-file = /var/run/mysqld/mysqld.pid
...   skip-external-locking
...   old_passwords = 1
...   skip-bdb
...   # we don't need ACID today
...   skip-innodb
... """
>>> config = configparser.ConfigParser(allow_no_value=True)
>>> config.read_string(sample_config)

>>> # Settings with values are treated as before:
>>> config["mysqld"]["user"]
'mysql'

>>> # Settings without values provide None:
>>> config["mysqld"]["skip-bdb"]

>>> # Settings which aren't specified still raise an error:
>>> config["mysqld"]["does-not-exist"]
Traceback (most recent call last):
  ...
KeyError: 'does-not-exist'
```

— *delimiters*, valeur par défaut : `('=', ':')`

Chaînes de caractères qui séparent les clés des valeurs à l'intérieur d'une section. La première occurrence d'une telle chaîne à l'intérieur d'une ligne est considérée comme un délimiteur. Cela signifie que les valeurs peuvent contenir certains des délimiteurs (mais pas les clés).

Voir aussi l'argument *space_around_delimiters* de la méthode `ConfigParser.write()`.

— *comment_prefixes* (préfixes de commentaire) — valeur par défaut : `('#', ';')`

— *inline_comment_prefixes* (préfixes de commentaire en ligne) — valeur par défaut : `('#', ';')`

Les préfixes de commentaire indiquent le début d'un commentaire valide au sein d'un fichier de configuration. Ils ne peuvent être utilisés qu'à l'emplacement d'une ligne vide (potentiellement indentée). En revanche, les

préfixes de commentaires en ligne peuvent être utilisés après n'importe quelle valeur valide (comme les noms des sections, les options et les lignes vides). Par défaut, les commentaires en ligne sont désactivés et les préfixes utilisés pour les commentaires à l'emplacement d'une ligne vide sont `'#'` et `';'`.

Modifié dans la version 3.2 : Les précédentes versions du module *configparser* se comportent comme en utilisant `comment_prefixes=('#',';')` et `inline_comment_prefixes=(';',)`.

Notez que les analyseurs ne prennent pas en charge l'échappement des préfixes de commentaires. Ainsi, l'utilisation de *inline_comment_prefixes* peut empêcher les utilisateurs de spécifier des valeurs qui contiennent des caractères utilisés comme préfixe de commentaire. Dans le doute, il est recommandé de ne pas utiliser *inline_comment_prefixes*. Dans tous les cas, la seule façon de stocker des préfixes de commentaires au début d'une valeur multi lignes est d'interpoler ceux-ci, par exemple :

```
>>> from configparser import ConfigParser, ExtendedInterpolation
>>> parser = ConfigParser(interpolation=ExtendedInterpolation())
>>> # the default BasicInterpolation could be used as well
>>> parser.read_string("""
... [DEFAULT]
... hash = #
...
... [hashes]
... shebang =
...   ${hash}!/usr/bin/env python
...   ${hash} -*- coding: utf-8 -*-
...
... extensions =
...   enabled_extension
...   another_extension
...   #disabled_by_comment
...   yet_another_extension
...
... interpolation not necessary = if # is not at line start
... even in multiline values = line #1
...   line #2
...   line #3
... """)
>>> print(parser['hashes']['shebang'])

#!/usr/bin/env python
# -*- coding: utf-8 -*-
>>> print(parser['hashes']['extensions'])

enabled_extension
another_extension
yet_another_extension
>>> print(parser['hashes']['interpolation not necessary'])
if # is not at line start
>>> print(parser['hashes']['even in multiline values'])
line #1
line #2
line #3
```

— *scrict*, valeur par défaut : `True`

When set to `True`, the parser will not allow for any section or option duplicates while reading from a single source (using *read_file()*, *read_string()* or *read_dict()*). It is recommended to use strict parsers in new applications.

Modifié dans la version 3.2 : Les versions précédentes du module *configparser* se comportent comme en utilisant `strict=False`.

— *empty_lines_in_values*, valeur par défaut : `True`

Du point de vue des analyseurs, les valeurs peuvent s'étendre sur plusieurs lignes à partir du moment où elles sont plus indentées que la clé qui les contient. Par défaut les analyseurs autorisent les lignes vides à faire partie de telles valeurs. Dans le même temps, les clés elles-mêmes peuvent être indentées de façon à rendre le fichier plus lisible. En conséquence, il est probable que l'utilisateur perde de vue la structure du fichier lorsque celui-ci devient long et complexe. Prenez par exemple :

```
[Section]
key = multiline
  value with a gotcha

 this = is still a part of the multiline value of 'key'
```

Ceci est particulièrement problématique si l'utilisateur a configuré son éditeur pour utiliser une police à chasse variable. C'est pourquoi il est conseillé de ne pas prendre en charge les valeurs avec des lignes vides, à moins que votre application en ait besoin. Dans ce cas, les lignes vides sont toujours interprétées comme séparant des clés. Dans l'exemple ci-dessus, cela produit deux clés : `key` et `this`.

— *default_section*, valeur par défaut : `configparser.DEFAULTSECT` (autrement dit : `"DEFAULT"`)

La convention permettant d'autoriser une section spéciale de valeurs par défaut pour d'autres sections ou à des fins d'interpolation est un concept puissant de cette bibliothèque, permettant aux utilisateurs de créer des configurations déclaratives complexes. Cette section est normalement appelée `"DEFAULT"`, mais elle peut être personnalisée pour pointer vers n'importe quel autre nom de section valide. Certaines autres valeurs typiques incluent : `"general"` ou `"common"`. Le nom fourni est utilisé pour reconnaître les sections par défaut lors de la lecture à partir de n'importe quelle source, et est utilisée lors de la réécriture de la configuration dans un fichier. Sa valeur actuelle peut être récupérée à l'aide de l'attribut `parser_instance.default_section`, et peut être modifiée au moment de l'exécution (c'est-à-dire pour convertir des fichiers d'un format à un autre).

— *interpolation*, valeur par défaut : `configparser.BasicInterpolation`

Le comportement d'interpolation peut être personnalisé en fournissant un gestionnaire personnalisé via l'argument *interpolation*. `None` peut être utilisé pour désactiver complètement l'interpolation. `ExtendedInterpolation()` fournit une variante plus avancée inspirée de `zc.buildout`. Plus d'informations sur le sujet dans la section *dédiée de la documentation*. `RowConfigParser` vaut `None` par défaut.

— *converters*, valeur par défaut : non défini

Les analyseurs de configuration fournissent des fonctions de récupération de valeurs qui effectuent la conversion par type. Par défaut, *`getint()`*, *`getfloat()`* et *`getboolean()`* sont implémentées. Si vous désirez d'autres fonctions de récupération de valeurs, vous pouvez les définir dans une sous-classe ou transmettre un dictionnaire où chaque clé est le nom du convertisseur et chaque valeur est une fonction implémentant ladite conversion. Par exemple, passer `{'decimal':decimal.Decimal}` ajouterait `getdecimal()` à la fois sur l'analyseur et sur tous les proxys de section. En d'autres termes, il sera possible d'écrire à la fois `parser_instance.getdecimal('section', 'key', fallback=0)` et `parser_instance['section'].getdecimal('key', 0)`.

Si le convertisseur doit accéder à l'état de l'analyseur, il peut être implémenté en tant que méthode sur une sous-classe d'analyseur de configuration. Si le nom de cette méthode commence par `get`, elle sera disponible sur tous les proxys de section, sous la forme compatible avec un dictionnaire (voir l'exemple `getdecimal()` ci-dessus).

Une personnalisation plus avancée peut être obtenue en remplaçant les valeurs par défaut de ces attributs de l'analyseur. Les valeurs par défaut sont définies sur les classes, elles peuvent donc être remplacées par des sous-classes ou par des attributs d'instance.

`ConfigParser.`**`BOOLEAN_STATES`**

Par défaut, la méthode *`getboolean()`* considère les valeurs suivantes comme vraies : `'1'`, `'yes'`, `'true'`, `'on'`, et les valeurs suivantes comme fausses : `'0'`, `'no'`, `'false'`, `'off'`. Vous pouvez changer ce comportement en spécifiant votre propre dictionnaire associant des chaînes de caractères à des valeurs booléennes. Par exemple :

```
>>> custom = configparser.ConfigParser()
>>> custom['section1'] = {'funky': 'nope'}
>>> custom['section1'].getboolean('funky')
```

(suite sur la page suivante)

(suite de la page précédente)

```
Traceback (most recent call last):
...
ValueError: Not a boolean: nope
>>> custom.BOOLEAN_STATES = {'sure': True, 'nope': False}
>>> custom['section1'].getboolean('funky')
False
```

Other typical Boolean pairs include `accept`/`reject` or `enabled`/`disabled`.

`ConfigParser.`**`optionxform`**(*option*)

This method transforms option names on every read, get, or set operation. The default converts the name to lowercase. This also means that when a configuration file gets written, all keys will be lowercase. Override this method if that's unsuitable. For example :

```
>>> config = """
... [Section1]
... Key = Value
...
... [Section2]
... AnotherKey = Value
... """
>>> typical = configparser.ConfigParser()
>>> typical.read_string(config)
>>> list(typical['Section1'].keys())
['key']
>>> list(typical['Section2'].keys())
['anotherkey']
>>> custom = configparser.RawConfigParser()
>>> custom.optionxform = lambda option: option
>>> custom.read_string(config)
>>> list(custom['Section1'].keys())
['Key']
>>> list(custom['Section2'].keys())
['AnotherKey']
```

Note

The optionxform function transforms option names to a canonical form. This should be an idempotent function : if the name is already in canonical form, it should be returned unchanged.

`ConfigParser.`**`SECTCRE`**

A compiled regular expression used to parse section headers. The default matches `[section]` to the name `"section"`. Whitespace is considered part of the section name, thus `[ larch ]` will be read as a section of name `" larch "`. Override this attribute if that's unsuitable. For example :

```
>>> import re
>>> config = """
... [Section 1]
... option = value
...
... [  Section 2  ]
... another = val
... """
>>> typical = configparser.ConfigParser()
>>> typical.read_string(config)
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> typical.sections()
['Section 1', '  Section 2  ']
>>> custom = configparser.ConfigParser()
>>> custom.SECTCRE = re.compile(r"\[ *(?P<header>[^]]+?) *\]")
>>> custom.read_string(config)
>>> custom.sections()
['Section 1', 'Section 2']
```

> **Note**
>
> While ConfigParser objects also use an `OPTCRE` attribute for recognizing option lines, it's not recommended to override it because that would interfere with constructor options *allow_no_value* and *delimiters*.

14.2.9 Legacy API Examples

Mainly because of backwards compatibility concerns, *`configparser`* provides also a legacy API with explicit `get`/`set` methods. While there are valid use cases for the methods outlined below, mapping protocol access is preferred for new projects. The legacy API is at times more advanced, low-level and downright counterintuitive.

An example of writing to a configuration file :

```
import configparser

config = configparser.RawConfigParser()

# Please note that using RawConfigParser's set functions, you can assign
# non-string values to keys internally, but will receive an error when
# attempting to write to a file or when you get it in non-raw mode. Setting
# values using the mapping protocol or ConfigParser's set() does not allow
# such assignments to take place.
config.add_section('Section1')
config.set('Section1', 'an_int', '15')
config.set('Section1', 'a_bool', 'true')
config.set('Section1', 'a_float', '3.1415')
config.set('Section1', 'baz', 'fun')
config.set('Section1', 'bar', 'Python')
config.set('Section1', 'foo', '%(bar)s is %(baz)s!')

# Writing our configuration file to 'example.cfg'
with open('example.cfg', 'w') as configfile:
    config.write(configfile)
```

An example of reading the configuration file again :

```
import configparser

config = configparser.RawConfigParser()
config.read('example.cfg')

# getfloat() raises an exception if the value is not a float
# getint() and getboolean() also do this for their respective types
a_float = config.getfloat('Section1', 'a_float')
an_int = config.getint('Section1', 'an_int')
print(a_float + an_int)

# Notice that the next output does not interpolate '%(bar)s' or '%(baz)s'.
```

(suite sur la page suivante)

(suite de la page précédente)

```
# This is because we are using a RawConfigParser().
if config.getboolean('Section1', 'a_bool'):
    print(config.get('Section1', 'foo'))
```

To get interpolation, use `ConfigParser` :

```
import configparser

cfg = configparser.ConfigParser()
cfg.read('example.cfg')

# Set the optional *raw* argument of get() to True if you wish to disable
# interpolation in a single get operation.
print(cfg.get('Section1', 'foo', raw=False))  # -> "Python is fun!"
print(cfg.get('Section1', 'foo', raw=True))   # -> "%(bar)s is %(baz)s!"

# The optional *vars* argument is a dict with members that will take
# precedence in interpolation.
print(cfg.get('Section1', 'foo', vars={'bar': 'Documentation',
                                       'baz': 'evil'}))

# The optional *fallback* argument can be used to provide a fallback value
print(cfg.get('Section1', 'foo'))
      # -> "Python is fun!"

print(cfg.get('Section1', 'foo', fallback='Monty is not.'))
      # -> "Python is fun!"

print(cfg.get('Section1', 'monster', fallback='No such things as monsters.'))
      # -> "No such things as monsters."

# A bare print(cfg.get('Section1', 'monster')) would raise NoOptionError
# but we can also use:

print(cfg.get('Section1', 'monster', fallback=None))
      # -> None
```

Default values are available in both types of ConfigParsers. They are used in interpolation if an option used is not defined elsewhere.

```
import configparser

# New instance with 'bar' and 'baz' defaulting to 'Life' and 'hard' each
config = configparser.ConfigParser({'bar': 'Life', 'baz': 'hard'})
config.read('example.cfg')

print(config.get('Section1', 'foo'))      # -> "Python is fun!"
config.remove_option('Section1', 'bar')
config.remove_option('Section1', 'baz')
print(config.get('Section1', 'foo'))      # -> "Life is hard!"
```

14.2.10 ConfigParser Objects

class configparser.**ConfigParser**(*defaults=None*, *dict_type=dict*, *allow_no_value=False*, *, *delimiters=('=', ':')*, *comment_prefixes=('#', ';')*, *inline_comment_prefixes=None*, *strict=True*, *empty_lines_in_values=True*, *default_section=configparser.DEFAULTSECT*, *interpolation=BasicInterpolation()*, *converters={}*, *allow_unnamed_section=False*)

The main configuration parser. When *defaults* is given, it is initialized into the dictionary of intrinsic defaults. When *dict_type* is given, it will be used to create the dictionary objects for the list of sections, for the options within a section, and for the default values.

When *delimiters* is given, it is used as the set of substrings that divide keys from values. When *comment_prefixes* is given, it will be used as the set of substrings that prefix comments in otherwise empty lines. Comments can be indented. When *inline_comment_prefixes* is given, it will be used as the set of substrings that prefix comments in non-empty lines.

When *strict* is `True` (the default), the parser won't allow for any section or option duplicates while reading from a single source (file, string or dictionary), raising `DuplicateSectionError` or `DuplicateOptionError`. When *empty_lines_in_values* is `False` (default : `True`), each empty line marks the end of an option. Otherwise, internal empty lines of a multiline option are kept as part of the value. When *allow_no_value* is `True` (default : `False`), options without values are accepted ; the value held for these is `None` and they are serialized without the trailing delimiter.

When *default_section* is given, it specifies the name for the special section holding default values for other sections and interpolation purposes (normally named `"DEFAULT"`). This value can be retrieved and changed at runtime using the `default_section` instance attribute. This won't re-evaluate an already parsed config file, but will be used when writing parsed settings to a new config file.

Interpolation behaviour may be customized by providing a custom handler through the *interpolation* argument. `None` can be used to turn off interpolation completely, `ExtendedInterpolation()` provides a more advanced variant inspired by `zc.buildout`. More on the subject in the *dedicated documentation section*.

All option names used in interpolation will be passed through the `optionxform()` method just like any other option name reference. For example, using the default implementation of `optionxform()` (which converts option names to lower case), the values `foo %(bar)s` and `foo %(BAR)s` are equivalent.

When *converters* is given, it should be a dictionary where each key represents the name of a type converter and each value is a callable implementing the conversion from string to the desired datatype. Every converter gets its own corresponding `get*()` method on the parser object and section proxies.

When *allow_unnamed_section* is `True` (default : `False`), the first section name can be omitted. See the *"Unnamed Sections" section*.

Il est possible de lire plusieurs configurations dans un seul objet `ConfigParser`, où la configuration ajoutée la plus récemment a la priorité la plus élevée. Toutes les clés en conflit sont prises de la configuration la plus récente tandis que les anciennes clés déjà existantes sont mises de côté. L'exemple suivant lit un fichier `override.ini` qui prendra le dessus sur toutes les clés en conflit avec le fichier `exemple.ini`.

```
[DEFAULT]
ServerAliveInterval = -1
```

```
>>> config_override = configparser.ConfigParser()
>>> config_override['DEFAULT'] = {'ServerAliveInterval': '-1'}
>>> with open('override.ini', 'w') as configfile:
...     config_override.write(configfile)
...
>>> config_override = configparser.ConfigParser()
>>> config_override.read(['example.ini', 'override.ini'])
['example.ini', 'override.ini']
>>> print(config_override.get('DEFAULT', 'ServerAliveInterval'))
-1
```

Modifié dans la version 3.1 : The default *dict_type* is `collections.OrderedDict`.

Modifié dans la version 3.2 : *allow_no_value*, *delimiters*, *comment_prefixes*, *strict*, *empty_lines_in_values*, *default_section* and *interpolation* were added.

Modifié dans la version 3.5 : The *converters* argument was added.

Modifié dans la version 3.7 : The *defaults* argument is read with `read_dict()`, providing consistent behavior across the parser : non-string keys and values are implicitly converted to strings.

Modifié dans la version 3.8 : The default *dict_type* is `dict`, since it now preserves insertion order.

Modifié dans la version 3.13 : Raise a `MultilineContinuationError` when *allow_no_value* is `True`, and a key without a value is continued with an indented line.

Modifié dans la version 3.13 : The *allow_unnamed_section* argument was added.

defaults()

Return a dictionary containing the instance-wide defaults.

sections()

Return a list of the sections available ; the *default section* is not included in the list.

add_section(*section*)

Add a section named *section* to the instance. If a section by the given name already exists, `DuplicateSectionError` is raised. If the *default section* name is passed, `ValueError` is raised. The name of the section must be a string ; if not, `TypeError` is raised.

Modifié dans la version 3.2 : Non-string section names raise `TypeError`.

has_section(*section*)

Indicates whether the named *section* is present in the configuration. The *default section* is not acknowledged.

options(*section*)

Return a list of options available in the specified *section*.

has_option(*section*, *option*)

If the given *section* exists, and contains the given *option*, return `True` ; otherwise return `False`. If the specified *section* is `None` or an empty string, DEFAULT is assumed.

read(*filenames*, *encoding=None*)

Attempt to read and parse an iterable of filenames, returning a list of filenames which were successfully parsed.

If *filenames* is a string, a `bytes` object or a *path-like object*, it is treated as a single filename. If a file named in *filenames* cannot be opened, that file will be ignored. This is designed so that you can specify an iterable of potential configuration file locations (for example, the current directory, the user's home directory, and some system-wide directory), and all existing configuration files in the iterable will be read.

If none of the named files exist, the `ConfigParser` instance will contain an empty dataset. An application which requires initial values to be loaded from a file should load the required file or files using `read_file()` before calling `read()` for any optional files :

```
import configparser, os

config = configparser.ConfigParser()
config.read_file(open('defaults.cfg'))
config.read(['site.cfg', os.path.expanduser('~/.myapp.cfg')],
            encoding='cp1250')
```

Modifié dans la version 3.2 : Added the *encoding* parameter. Previously, all files were read using the default encoding for `open()`.

Modifié dans la version 3.6.1 : The *filenames* parameter accepts a *path-like object*.

Modifié dans la version 3.7 : The *filenames* parameter accepts a `bytes` object.

read_file(*f*, *source=None*)

Read and parse configuration data from *f* which must be an iterable yielding Unicode strings (for example files opened in text mode).

Optional argument *source* specifies the name of the file being read. If not given and *f* has a `name` attribute, that is used for *source*; the default is `'<???>'`.

Ajouté dans la version 3.2 : Replaces `readfp()`.

read_string(*string*, *source='<string>'*)

Parse configuration data from a string.

Optional argument *source* specifies a context-specific name of the string passed. If not given, `'<string>'` is used. This should commonly be a filesystem path or a URL.

Ajouté dans la version 3.2.

read_dict(*dictionary*, *source='<dict>'*)

Load configuration from any object that provides a dict-like `items()` method. Keys are section names, values are dictionaries with keys and values that should be present in the section. If the used dictionary type preserves order, sections and their keys will be added in order. Values are automatically converted to strings.

Optional argument *source* specifies a context-specific name of the dictionary passed. If not given, `<dict>` is used.

This method can be used to copy state between parsers.

Ajouté dans la version 3.2.

get(*section*, *option*, *, *raw=False*, *vars=None*[, *fallback*])

Get an *option* value for the named *section*. If *vars* is provided, it must be a dictionary. The *option* is looked up in *vars* (if provided), *section*, and in *DEFAULTSECT* in that order. If the key is not found and *fallback* is provided, it is used as a fallback value. `None` can be provided as a *fallback* value.

All the `'%'` interpolations are expanded in the return values, unless the *raw* argument is true. Values for interpolation keys are looked up in the same manner as the option.

Modifié dans la version 3.2 : Arguments *raw*, *vars* and *fallback* are keyword only to protect users from trying to use the third argument as the *fallback* fallback (especially when using the mapping protocol).

getint(*section*, *option*, *, *raw=False*, *vars=None*[, *fallback*])

A convenience method which coerces the *option* in the specified *section* to an integer. See `get()` for explanation of *raw*, *vars* and *fallback*.

getfloat(*section*, *option*, *, *raw=False*, *vars=None*[, *fallback*])

A convenience method which coerces the *option* in the specified *section* to a floating-point number. See `get()` for explanation of *raw*, *vars* and *fallback*.

getboolean(*section*, *option*, *, *raw=False*, *vars=None*[, *fallback*])

A convenience method which coerces the *option* in the specified *section* to a Boolean value. Note that the accepted values for the option are `'1'`, `'yes'`, `'true'`, and `'on'`, which cause this method to return `True`, and `'0'`, `'no'`, `'false'`, and `'off'`, which cause it to return `False`. These string values are checked in a case-insensitive manner. Any other value will cause it to raise `ValueError`. See `get()` for explanation of *raw*, *vars* and *fallback*.

items(*raw=False*, *vars=None*)

items(*section*, *raw=False*, *vars=None*)

When *section* is not given, return a list of *section_name*, *section_proxy* pairs, including DEFAULTSECT.

Otherwise, return a list of *name*, *value* pairs for the options in the given *section*. Optional arguments have the same meaning as for the `get()` method.

Modifié dans la version 3.8 : Items present in *vars* no longer appear in the result. The previous behaviour

mixed actual parser options with variables provided for interpolation.

set(*section*, *option*, *value*)

If the given section exists, set the given option to the specified value; otherwise raise `NoSectionError`. *option* and *value* must be strings; if not, `TypeError` is raised.

write(*fileobject*, *space_around_delimiters=True*)

Write a representation of the configuration to the specified *file object*, which must be opened in text mode (accepting strings). This representation can be parsed by a future `read()` call. If *space_around_delimiters* is true, delimiters between keys and values are surrounded by spaces.

> **Note**
>
> Comments in the original configuration file are not preserved when writing the configuration back. What is considered a comment, depends on the given values for *comment_prefix* and *inline_comment_prefix*.

remove_option(*section*, *option*)

Remove the specified *option* from the specified *section*. If the section does not exist, raise `NoSectionError`. If the option existed to be removed, return `True`; otherwise return `False`.

remove_section(*section*)

Remove the specified *section* from the configuration. If the section in fact existed, return `True`. Otherwise return `False`.

optionxform(*option*)

Transforms the option name *option* as found in an input file or as passed in by client code to the form that should be used in the internal structures. The default implementation returns a lower-case version of *option*; subclasses may override this or client code can set an attribute of this name on instances to affect this behavior.

You don't need to subclass the parser to use this method, you can also set it on an instance, to a function that takes a string argument and returns a string. Setting it to `str`, for example, would make option names case sensitive :

```
cfgparser = ConfigParser()
cfgparser.optionxform = str
```

Note that when reading configuration files, whitespace around the option names is stripped before `optionxform()` is called.

configparser.**UNNAMED_SECTION**

A special object representing a section name used to reference the unnamed section (see *Unnamed Sections*).

configparser.**MAX_INTERPOLATION_DEPTH**

The maximum depth for recursive interpolation for `get()` when the *raw* parameter is false. This is relevant only when the default *interpolation* is used.

14.2.11 RawConfigParser Objects

class configparser.**RawConfigParser**(*defaults=None*, *dict_type=dict*, *allow_no_value=False*, ***, *delimiters=('=', ':')*, *comment_prefixes=('#', ';')*, *inline_comment_prefixes=None*, *strict=True*, *empty_lines_in_values=True*, *default_section=configparser.DEFAULTSECT*, *interpolation=BasicInterpolation()*, *converters={}*, *allow_unnamed_section=False*)

Legacy variant of the `ConfigParser`. It has interpolation disabled by default and allows for non-string section names, option names, and values via its unsafe `add_section` and `set` methods, as well as the legacy `defaults=` keyword argument handling.

Modifié dans la version 3.2 : *allow_no_value*, *delimiters*, *comment_prefixes*, *strict*, *empty_lines_in_values*, *default_section* and *interpolation* were added.

Modifié dans la version 3.5 : The *converters* argument was added.

Modifié dans la version 3.8 : The default *dict_type* is `dict`, since it now preserves insertion order.

Modifié dans la version 3.13 : The *allow_unnamed_section* argument was added.

> **Note**
>
> Consider using `ConfigParser` instead which checks types of the values to be stored internally. If you don't want interpolation, you can use `ConfigParser(interpolation=None)`.

add_section(*section*)

Add a section named *section* or `UNNAMED_SECTION` to the instance.

If the given section already exists, `DuplicateSectionError` is raised. If the *default section* name is passed, `ValueError` is raised. If `UNNAMED_SECTION` is passed and support is disabled, `UnnamedSectionDisabledError` is raised.

Type of *section* is not checked which lets users create non-string named sections. This behaviour is unsupported and may cause internal errors.

Modifié dans la version 3.14 : Added support for `UNNAMED_SECTION`.

set(*section*, *option*, *value*)

If the given section exists, set the given option to the specified value ; otherwise raise `NoSectionError`. While it is possible to use `RawConfigParser` (or `ConfigParser` with *raw* parameters set to true) for *internal* storage of non-string values, full functionality (including interpolation and output to files) can only be achieved using string values.

This method lets users assign non-string values to keys internally. This behaviour is unsupported and will cause errors when attempting to write to a file or get it in non-raw mode. **Use the mapping protocol API** which does not allow such assignments to take place.

14.2.12 Exceptions

exception configparser.**Error**

Base class for all other `configparser` exceptions.

exception configparser.**NoSectionError**

Exception raised when a specified section is not found.

exception configparser.**DuplicateSectionError**

Exception raised if `add_section()` is called with the name of a section that is already present or in strict parsers when a section if found more than once in a single input file, string or dictionary.

Modifié dans la version 3.2 : Added the optional *source* and *lineno* attributes and parameters to `__init__()`.

exception configparser.**DuplicateOptionError**

Exception raised by strict parsers if a single option appears twice during reading from a single file, string or dictionary. This catches misspellings and case sensitivity-related errors, e.g. a dictionary may have two keys representing the same case-insensitive configuration key.

exception configparser.**NoOptionError**

Exception raised when a specified option is not found in the specified section.

exception configparser.**InterpolationError**

Base class for exceptions raised when problems occur performing string interpolation.

exception configparser.**InterpolationDepthError**

Exception raised when string interpolation cannot be completed because the number of iterations exceeds *MAX_INTERPOLATION_DEPTH*. Subclass of *InterpolationError*.

exception configparser.**InterpolationMissingOptionError**

Exception raised when an option referenced from a value does not exist. Subclass of *InterpolationError*.

exception configparser.**InterpolationSyntaxError**

Exception raised when the source text into which substitutions are made does not conform to the required syntax. Subclass of *InterpolationError*.

exception configparser.**MissingSectionHeaderError**

Exception raised when attempting to parse a file which has no section headers.

exception configparser.**ParsingError**

Exception raised when errors occur attempting to parse a file.

Modifié dans la version 3.12 : The `filename` attribute and `__init__()` constructor argument were removed. They have been available using the name `source` since 3.2.

exception configparser.**MultilineContinuationError**

Exception raised when a key without a corresponding value is continued with an indented line.

Ajouté dans la version 3.13.

exception configparser.**UnnamedSectionDisabledError**

Exception raised when attempting to use the *UNNAMED_SECTION* without enabling it.

Ajouté dans la version 3.14.

Notes

14.3 `tomllib` --- Parse TOML files

Ajouté dans la version 3.11.

Source code : Lib/tomllib

This module provides an interface for parsing TOML 1.0.0 (Tom's Obvious Minimal Language, https ://toml.io). This module does not support writing TOML.

> **Voir aussi**
>
> The Tomli-W package is a TOML writer that can be used in conjunction with this module, providing a write API familiar to users of the standard library *marshal* and *pickle* modules.

> **Voir aussi**
>
> The TOML Kit package is a style-preserving TOML library with both read and write capability. It is a recommended replacement for this module for editing already existing TOML files.

Ce module définit les fonctions suivantes :

tomllib.**load**(*fp*, /, *, *parse_float=float*)

Read a TOML file. The first argument should be a readable and binary file object. Return a *dict*. Convert TOML types to Python using this *conversion table*.

parse_float will be called with the string of every TOML float to be decoded. By default, this is equivalent to `float(num_str)`. This can be used to use another datatype or parser for TOML floats (e.g. `decimal.Decimal`). The callable must not return a `dict` or a `list`, else a `ValueError` is raised.

A `TOMLDecodeError` will be raised on an invalid TOML document.

`tomllib.loads`(*s*, /, *, *parse_float=float*)

Load TOML from a `str` object. Return a `dict`. Convert TOML types to Python using this *conversion table*. The *parse_float* argument has the same meaning as in `load()`.

A `TOMLDecodeError` will be raised on an invalid TOML document.

The following exceptions are available :

exception `tomllib.TOMLDecodeError`(*msg*, *doc*, *pos*)

Subclass of `ValueError` with the following additional attributes :

msg

The unformatted error message.

doc

The TOML document being parsed.

pos

The index of *doc* where parsing failed.

lineno

The line corresponding to *pos*.

colno

The column corresponding to *pos*.

Modifié dans la version 3.14 : Added the *msg*, *doc* and *pos* parameters. Added the `msg`, `doc`, `pos`, `lineno` and `colno` attributes.

Obsolète depuis la version 3.14 : Passing free-form positional arguments is deprecated.

14.3.1 Exemples

Parsing a TOML file :

```
import tomllib

with open("pyproject.toml", "rb") as f:
    data = tomllib.load(f)
```

Parsing a TOML string :

```
import tomllib

toml_str = """
python-version = "3.11.0"
python-implementation = "CPython"
"""

data = tomllib.loads(toml_str)
```

14.3.2 Conversion Table

TOML	Python
TOML document	*dict*
string	*str*
int (entier)	*int*
float (nombre à virgule flottante)	float (configurable with *parse_float*)
boolean	*bool* (booléen)
offset date-time	datetime.datetime (`tzinfo` attribute set to an instance of `datetime.timezone`)
local date-time	datetime.datetime (`tzinfo` attribute set to `None`)
local date	datetime.date
local time	datetime.time
array	liste
table	*dict*
inline table	*dict*
array of tables	list of dicts

14.4 `netrc` --- netrc file processing

Code source : Lib/netrc.py

La classe `netrc` analyse et encapsule le format de fichier *netrc* utilisé par le programme Unix **ftp** et d'autres clients FTP.

class netrc.**netrc**([*file*])

Une instance de `netrc` ou une instance de sous-classe encapsule les données à partir d'un fichier *netrc*. L'argument d'initialisation, s'il est présent, précise le fichier à analyser. Si aucun argument n'est donné, le fichier `.netrc` dans le répertoire d'accueil de l'utilisateur -- déterminé par `os.path.expanduser()` -- est lu. Sinon, l'exception `FileNotFoundError` sera levée. Les erreurs d'analyse lèveront `NetrcParseError` avec les informations de diagnostic, y compris le nom de fichier, le numéro de ligne, et le lexème. Si aucun argument n'est spécifié dans un système POSIX, la présence de mots de passe dans le fichier `.netrc` lèvera `NetrcParseError` si la propriété du fichier ou les permissions ne sont pas sécurisées (propriété d'un utilisateur autre que l'utilisateur exécutant le processus ou accessible en lecture ou en écriture par n'importe quel autre utilisateur). Le niveau de sécurité offert est ainsi équivalent à celui de ftp et d'autres programmes utilisant `netrc`.

Modifié dans la version 3.4 : Ajout de la vérification d'autorisations POSIX.

Modifié dans la version 3.7 : `os.path.expanduser()` est utilisée pour trouver l'emplacement du fichier `netrc` lorsque *file* n'est pas passé en tant qu'argument.

Modifié dans la version 3.10 : `netrc` try UTF-8 encoding before using locale specific encoding. The entry in the netrc file no longer needs to contain all tokens. The missing tokens' value default to an empty string. All the tokens and their values now can contain arbitrary characters, like whitespace and non-ASCII characters. If the login name is anonymous, it won't trigger the security check.

exception netrc.**NetrcParseError**

Exception raised by the `netrc` class when syntactical errors are encountered in source text. Instances of this exception provide three interesting attributes :

msg

Textual explanation of the error.

filename

The name of the source file.

lineno
The line number on which the error was found.

14.4.1 Objets *netrc*

Une instance `netrc` a les méthodes suivantes :

`netrc.`**`authenticators`**(*host*)
Renvoie un triplet `(login, account, password)` pour s'authentifier auprès de l'hôte *host*. Si le fichier *netrc* ne contient pas d'entrée pour l'hôte donné, renvoie le triplet associé à l'entrée par défaut. Si aucun hôte correspondant ni aucune entrée par défaut n'est disponible, renvoie `None`.

`netrc.`**`__repr__`**()
Déverse les données de la classe sous forme de chaîne dans le format d'un fichier *netrc*. (Ceci ignore les commentaires et peut réorganiser les entrées).

Les instances de `netrc` ont des variables d'instance publiques :

`netrc.`**`hosts`**
Dictionnaire faisant correspondre les noms d'hôtes dans des triplets `(login, account, password)`. L'entrée par défaut, le cas échéant, est représentée en tant que pseudo-hôte par ce nom.

`netrc.`**`macros`**
Dictionnaire faisant correspondre les noms de macro en listes de chaînes.

14.5 `plistlib` --- Generate and parse Apple `.plist` files

Code source : Lib/plistlib.py

This module provides an interface for reading and writing the "property list" files used by Apple, primarily on macOS and iOS. This module supports both binary and XML plist files.

The property list (`.plist`) file format is a simple serialization supporting basic object types, like dictionaries, lists, numbers and strings. Usually the top level object is a dictionary.

To write out and to parse a plist file, use the `dump()` and `load()` functions.

To work with plist data in bytes or string objects, use `dumps()` and `loads()`.

Values can be strings, integers, floats, booleans, tuples, lists, dictionaries (but only with string keys), `bytes`, `bytearray` or `datetime.datetime` objects.

Modifié dans la version 3.4 : New API, old API deprecated. Support for binary format plists added.

Modifié dans la version 3.8 : Support added for reading and writing `UID` tokens in binary plists as used by NSKeyedArchiver and NSKeyedUnarchiver.

Modifié dans la version 3.9 : Old API removed.

Voir aussi

PList manual page
Apple's documentation of the file format.

Ce module définit les fonctions suivantes :

`plistlib.`**`load`**(*fp*, *, *fmt=None*, *dict_type=dict*, *aware_datetime=False*)
Read a plist file. *fp* should be a readable and binary file object. Return the unpacked root object (which usually is a dictionary).

The *fmt* is the format of the file and the following values are valid :

— `None` : Autodetect the file format
— `FMT_XML` : XML file format
— `FMT_BINARY` : Binary plist format
The *dict_type* is the type used for dictionaries that are read from the plist file.

When *aware_datetime* is true, fields with type `datetime.datetime` will be created as *aware object*, with `tzinfo` as `datetime.UTC`.

XML data for the `FMT_XML` format is parsed using the Expat parser from `xml.parsers.expat` -- see its documentation for possible exceptions on ill-formed XML. Unknown elements will simply be ignored by the plist parser.

The parser for the binary format raises `InvalidFileException` when the file cannot be parsed.

Ajouté dans la version 3.4.

Modifié dans la version 3.13 : The keyword-only parameter *aware_datetime* has been added.

`plistlib.loads`(*data*, *, *fmt=None*, *dict_type=dict*, *aware_datetime=False*)

Load a plist from a bytes or string object. See `load()` for an explanation of the keyword arguments.

Ajouté dans la version 3.4.

Modifié dans la version 3.13 : *data* can be a string when *fmt* equals `FMT_XML`.

`plistlib.dump`(*value*, *fp*, *, *fmt=FMT_XML*, *sort_keys=True*, *skipkeys=False*, *aware_datetime=False*)

Write *value* to a plist file. *fp* should be a writable, binary file object.

The *fmt* argument specifies the format of the plist file and can be one of the following values :
— `FMT_XML` : XML formatted plist file
— `FMT_BINARY` : Binary formatted plist file
When *sort_keys* is true (the default) the keys for dictionaries will be written to the plist in sorted order, otherwise they will be written in the iteration order of the dictionary.

When *skipkeys* is false (the default) the function raises `TypeError` when a key of a dictionary is not a string, otherwise such keys are skipped.

When *aware_datetime* is true and any field with type `datetime.datetime` is set as an *aware object*, it will convert to UTC timezone before writing it.

A `TypeError` will be raised if the object is of an unsupported type or a container that contains objects of unsupported types.

An `OverflowError` will be raised for integer values that cannot be represented in (binary) plist files.

Ajouté dans la version 3.4.

Modifié dans la version 3.13 : The keyword-only parameter *aware_datetime* has been added.

`plistlib.dumps`(*value*, *, *fmt=FMT_XML*, *sort_keys=True*, *skipkeys=False*, *aware_datetime=False*)

Return *value* as a plist-formatted bytes object. See the documentation for `dump()` for an explanation of the keyword arguments of this function.

Ajouté dans la version 3.4.

The following classes are available :

class `plistlib.UID`(*data*)

Wraps an `int`. This is used when reading or writing NSKeyedArchiver encoded data, which contains UID (see PList manual).

It has one attribute, `data`, which can be used to retrieve the int value of the UID. `data` must be in the range `0 <= data < 2**64`.

Ajouté dans la version 3.8.

The following constants are available :

plistlib.**FMT_XML**

The XML format for plist files.

Ajouté dans la version 3.4.

plistlib.**FMT_BINARY**

The binary format for plist files

Ajouté dans la version 3.4.

14.5.1 Exemples

Generating a plist :

```
import datetime
import plistlib

pl = dict(
    aString = "Doodah",
    aList = ["A", "B", 12, 32.1, [1, 2, 3]],
    aFloat = 0.1,
    anInt = 728,
    aDict = dict(
        anotherString = "<hello & hi there!>",
        aThirdString = "M\xe4ssig, Ma\xdf",
        aTrueValue = True,
        aFalseValue = False,
    ),
    someData = b"<binary gunk>",
    someMoreData = b"<lots of binary gunk>" * 10,
    aDate = datetime.datetime.now()
)
print(plistlib.dumps(pl).decode())
```

Parsing a plist :

```
import plistlib

plist = b"""<plist version="1.0">
<dict>
    <key>foo</key>
    <string>bar</string>
</dict>
</plist>"""
pl = plistlib.loads(plist)
print(pl["foo"])
```

CHAPITRE 15

Service de cryptographie

The modules described in this chapter implement various algorithms of a cryptographic nature. They are available at the discretion of the installation. Here's an overview :

15.1 `hashlib` --- Algorithmes de hachage sécurisés et synthèse de messages

Code source : Lib/hashlib.py

Ce module implémente une interface commune à différents algorithmes de hachage sécurisés et de synthèse de messages. Sont inclus les algorithmes standards FIPS de hachage SHA1, SHA224, SHA256, SHA384, et SHA512 (définis dans le le standard FIPS 180-4), la série SHA-3 (définis dans le standard FIPS 202), ainsi que l'algorithme MD5 de RSA (défini par la **RFC 1321**). Les termes « algorithmes de hachage sécurisé » et « algorithme de synthèse de message » sont interchangeables. Les anciens algorithmes étaient appelés « algorithmes de synthèse de messages ». Le terme moderne est « algorithme de hachage sécurisé ».

> **Note**
>
> Si vous préférez utiliser les fonctions de hachage *adler32* ou *crc32*, elles sont disponibles dans le module `zlib`.

15.1.1 Algorithmes de hachage

Il y a un constructeur nommé selon chaque type de calcul d'empreinte (*hash*). Tous renvoient un objet haché avec la même interface. Par exemple : utilisez `sha256()` pour créer un objet de calcul d'empreinte avec l'algorithme SHA-256. Vous pouvez maintenant utiliser cet objet avec des *objets octets-compatibles* (normalement des `bytes`) en utilisant la méthode `update`. À tout moment vous pouvez demander l'empreinte (aussi appelée le *digest* ou le condensat) de la concaténation des données fournies en utilisant les méthodes `digest()` ou `hexdigest()`.

pour de meilleures performances avec de multiples fils d'exécution, le *GIL* Python est relâché pour des données dont la taille est supérieure à 2047 octets lors de leur création ou leur mise à jour (méthode `.update`.).

Les constructeurs pour les algorithmes de hachage qui sont toujours présents dans ce module sont `sha1()`, `sha224()`, `sha256()`, `sha384()`, `sha512()`, `sha3_224()`, `sha3_256()`, `sha3_384()`, `sha3_512()`, `shake_128()`, `shake_256()`, `blake2b()`, et `blake2s()`. `md5()` est normalement disponible aussi, mais il peut

être manquant ou bloqué si vous utilisez une forme rare de Python « conforme FIPS ». Cette liste correspond au résultat de la fonction `algorithms_guaranteed`.

Additional algorithms may also be available if your Python distribution's `hashlib` was linked against a build of OpenSSL that provides others. Others *are not guaranteed available* on all installations and will only be accessible by name via `new()`. See `algorithms_available`.

> **Avertissement**
>
> Certains algorithmes ont des faiblesses connues relatives à la collision (MD5 et SHA1 inclus). Se référer à *Fonction de hachage cryptographique* et à la section Voir aussi à la fin du document.

Ajouté dans la version 3.6 : les constructeurs SHA3 (Keccak) et SHAKE `sha3_224()`, `sha3_256()`, `sha3_384()`, `sha3_512()`, `shake_128()`, `shake_256()` ont été ajoutés. `blake2b()` et `blake2s()` ont été ajoutés. Modifié dans la version 3.9 : tous les constructeurs de *hashlib* prennent un argument mot-clé uniquement *usedforsecurity* avec la valeur par défaut `True`. Une valeur fausse permet l'utilisation d'algorithmes de hachage non sécurisés et bloqués dans des environnements restreints. `False` indique que l'algorithme de hachage ne sera pas utilisé dans un contexte de sécurité, par exemple en tant que fonction de compression à sens unique non cryptographique.

Modifié dans la version 3.9 : *Hashlib* utilise désormais SHA3 et SHAKE d'OpenSSL s'il les fournit.

Modifié dans la version 3.12 : For any of the MD5, SHA1, SHA2, or SHA3 algorithms that the linked OpenSSL does not provide we fall back to a verified implementation from the HACL* project.

15.1.2 Utilisation

To obtain the digest of the byte string `b"Nobody inspects the spammish repetition"` :

```
>>> import hashlib
>>> m = hashlib.sha256()
>>> m.update(b"Nobody inspects")
>>> m.update(b" the spammish repetition")
>>> m.digest()
b'\x03\x1e\xdd}Ae\x15\x93\xc5\xfe\\\x00o\xa5u+7\xfd\xdf\xf7\xbcN\x84:\xa6\xaf\x0c\
↪x95\x0fK\x94\x06'
>>> m.hexdigest()
'031edd7d41651593c5fe5c006fa5752b37fddff7bc4e843aa6af0c950f4b9406'
```

En plus condensé :

```
>>> hashlib.sha256(b"Nobody inspects the spammish repetition").hexdigest()
'031edd7d41651593c5fe5c006fa5752b37fddff7bc4e843aa6af0c950f4b9406'
```

15.1.3 Constructeurs

`hashlib.new`(*name*, [*data*,]*, *usedforsecurity=True*)

Is a generic constructor that takes the string *name* of the desired algorithm as its first parameter. It also exists to allow access to the above listed hashes as well as any other algorithms that your OpenSSL library may offer.

Using `new()` with an algorithm name :

```
>>> h = hashlib.new('sha256')
>>> h.update(b"Nobody inspects the spammish repetition")
>>> h.hexdigest()
'031edd7d41651593c5fe5c006fa5752b37fddff7bc4e843aa6af0c950f4b9406'
```

`hashlib.md5`([*data*,]*, *usedforsecurity=True*)

hashlib.**sha1**([*data*,]*, *usedforsecurity=True*)

hashlib.**sha224**([*data*,]*, *usedforsecurity=True*)

hashlib.**sha256**([*data*,]*, *usedforsecurity=True*)

hashlib.**sha384**([*data*,]*, *usedforsecurity=True*)

hashlib.**sha512**([*data*,]*, *usedforsecurity=True*)

hashlib.**sha3_224**([*data*,]*, *usedforsecurity=True*)

hashlib.**sha3_256**([*data*,]*, *usedforsecurity=True*)

hashlib.**sha3_384**([*data*,]*, *usedforsecurity=True*)

hashlib.**sha3_512**([*data*,]*, *usedforsecurity=True*)

Named constructors such as these are faster than passing an algorithm name to `new()`.

15.1.4 Attributes

Hashlib fournit les constantes de module suivantes :

hashlib.**algorithms_guaranteed**

Ensemble contenant les noms des algorithmes de hachage garantis d'être implémentés par ce module sur toutes les plate-formes. Notez que *md5* est dans cette liste bien que certains éditeurs diffusent une implémentation Python dont la bibliothèque « compatible FIPS » le bloque.

Ajouté dans la version 3.2.

hashlib.**algorithms_available**

Ensemble contenant les noms des algorithmes de hachage disponibles dans l'interpréteur Python. Ces noms sont reconnus lorsqu'ils sont passés à la fonction `new()`. `algorithms_guaranteed` est toujours un sous-ensemble. Le même algorithme peut apparaître plusieurs fois dans cet ensemble sous un nom différent (grâce à OpenSSL).

Ajouté dans la version 3.2.

15.1.5 Objets de calcul d'empreinte

Les valeurs suivantes sont fournies en tant qu'attributs constants des objets de calcul d'empreintes renvoyés par les constructeurs :

hash.**digest_size**

La taille du *hash* résultant en octets.

hash.**block_size**

La taille interne d'un bloc de l'algorithme de hachage en octets.

L'objet de hachage possède les attributs suivants :

hash.**name**

Le nom canonique de cette fonction de hachage, toujours en minuscule et que vous pouvez toujours transmettre à la fonction `new()` pour créer un autre objet de calcul d'empreinte de ce type.

Modifié dans la version 3.4 : l'attribut *name* est présent dans CPython depuis sa création, mais n'était pas spécifié formellement jusqu'à Python 3.4, il peut ne pas exister sur certaines plate-formes.

L'objet de calcul d'empreinte possède les méthodes suivantes :

`hash.update(data)`

Met à jour l'objet de hachage avec *bytes-like object*. Les appels répétés sont équivalents à un simple appel avec la concaténation de tous les arguments : `m.update(a); m.update(b)` est équivalent à `m.update(a+b)`.

`hash.digest()`

Renvoie l'empreinte (le *hash*) des données passées à la méthode `update()`. C'est un objet de type *bytes* de taille `digest_size` qui contient des octets dans l'intervalle 0 à 255.

`hash.hexdigest()`

Comme la méthode `digest()` sauf que l'empreinte renvoyée est une chaîne de caractères dont la longueur est double, contenant seulement des chiffres hexadécimaux. Elle peut être utilisée pour échanger sans risque des valeurs dans les *e-mails* ou dans les environnements non binaires.

`hash.copy()`

Renvoie une copie ("clone") de l'objet de calcul de hachage. Cela peut être utilisé pour calculer efficacement des empreintes de données qui commencent par la même sous-chaine.

15.1.6 Empreintes de messages de taille variable SHAKE

`hashlib.shake_128([data, ]*, usedforsecurity=True)`

`hashlib.shake_256([data, ]*, usedforsecurity=True)`

Les algorithmes `shake_128()` et `shake_256()` produisent des empreintes (ou condensats) de longueur variable avec des `longueurs_en_bits // 2` jusqu'à 128 ou 256 bits. Leurs méthodes de calcul d'empreinte requièrent une longueur. Les longueurs maximales ne sont pas limitées par l'algorithme SHAKE.

`shake.digest(length)`

Renvoie l'empreinte des données passées à la méthode `update()`. C'est un objet de type *bytes* de taille *length* qui contient des octets dans l'intervalle 0 à 255.

`shake.hexdigest(length)`

Comme la méthode `digest()` sauf que l'empreinte renvoyée est une chaîne de caractères dont la longueur est double, contenant seulement des chiffres hexadécimaux. Elle peut être utilisée pour échanger sans risque des valeurs dans les *e-mails* ou dans les environnements non binaires.

Exemple :

```
>>> h = hashlib.shake_256(b'Nobody inspects the spammish repetition')
>>> h.hexdigest(20)
'44709d6fcb83d92a76dcb0b668c98e1b1d3dafe7'
```

15.1.7 Calcul d'empreinte (ou hachage) de fichiers

Le module *hashlib* fournit une fonction utilitaire pour calculer efficacement l'empreinte (aussi appelée condensat) d'un fichier ou d'un objet fichier-compatible.

`hashlib.file_digest(fileobj, digest, /)`

Renvoie un objet de calcul d'empreinte qui a été mis à jour avec le contenu de *fileobj*.

fileobj doit être un objet fichier-compatible ouvert en lecture en mode binaire. Sont acceptés les instances d'objets fichiers produites par la fonction native `open()`, `BytesIO`, les objets connecteurs produits par `socket.socket.makefile()` et similaires. La fonction peut court-circuiter les entrées-sorties de Python et utiliser directement le descripteur de fichier de `fileno()`. Vous devez supposer que *fileobj* est dans un état inconnu après le retour de cette fonction ou si elle a levé une exception. C'est à l'appelant de fermer *fileobj*.

digest doit être un nom d'algorithme de hachage fourni en tant que *str*, un constructeur de hachage ou un appelable qui renvoie un objet de hachage.

Exemple :

```
>>> import io, hashlib, hmac
>>> with open(hashlib.__file__, "rb") as f:
...     digest = hashlib.file_digest(f, "sha256")
...
>>> digest.hexdigest()
'...'
```

```
>>> buf = io.BytesIO(b"somedata")
>>> mac1 = hmac.HMAC(b"key", digestmod=hashlib.sha512)
>>> digest = hashlib.file_digest(buf, lambda: mac1)
```

```
>>> digest is mac1
True
>>> mac2 = hmac.HMAC(b"key", b"somedata", digestmod=hashlib.sha512)
>>> mac1.digest() == mac2.digest()
True
```

Ajouté dans la version 3.11.

15.1.8 Dérivation de clé

Les algorithmes de dérivation de clés et d'étirement de clés sont conçus pour le hachage sécurisé de mots de passe. Des algorithmes naïfs comme `sha1(password)` ne sont pas résistants aux attaques par force brute. Une bonne fonction de hachage doit être paramétrable, lente et inclure un salage.

`hashlib.`**`pbkdf2_hmac`**(*hash_name*, *password*, *salt*, *iterations*, *dklen=None*)

La fonction fournit une fonction de dérivation conforme à PKCS#5 (*Public Key Cryptographic Standards #5 v2.0*). Elle utilise HMAC comme fonction de génération d'un pseudo-aléa.

La chaîne de caractères *hash_name* est le nom de l'algorithme de hachage désiré pour le HMAC, par exemple `"sha1"` ou `"sha256"`. *password* et *salt* sont interprétés comme des tampons d'octets. Les applications et bibliothèques doivent limiter *password* à une longueur raisonnable (comme `1024`). *salt* doit être de 16 octets ou plus et provenir d'une source correcte, e.g. `os.urandom()`.

Le nombre d'*itérations* doit être choisi en fonction de l'algorithme de hachage et de la puissance de calcul. À partir de 2022, des centaines de milliers d'itérations de SHA-256 sont suggérées. Pour savoir pourquoi et comment choisir ce qui convient le mieux à votre application, lisez *l'annexe A.2.2* du NIST-SP-800-132 (ressource en anglais). Les réponses contenues dans la question sur le site *stackexchange* nombre d'itérations recommandées lorsqu'on utilise pbkdf2 (page en anglais) expliquent tout en détail.

dklen est la longueur de la clé dérivée. Si *dklen* vaut `None` alors Python utilise la taille du condensat produit par l'algorithme de hachage *hash_name*, par exemple 64 pour SHA-512.

```
>>> from hashlib import pbkdf2_hmac
>>> our_app_iters = 500_000  # Application specific, read above.
>>> dk = pbkdf2_hmac('sha256', b'password', b'bad salt' * 2, our_app_iters)
>>> dk.hex()
'15530bba69924174860db778f2c6f8104d3aaf9d26241840c8c4a641c8d000a9'
```

Function only available when Python is compiled with OpenSSL.

Ajouté dans la version 3.4.

Modifié dans la version 3.12 : Function now only available when Python is built with OpenSSL. The slow pure Python implementation has been removed.

`hashlib.`**`scrypt`**(*password*, ***, *salt*, *n*, *r*, *p*, *maxmem=0*, *dklen=64*)

La fonction fournit la fonction de dérivation de clé *scrypt* comme définie dans la **RFC 7914**.

password et *salt* doivent être des *objets octets-compatibles*. Les applications et bibliothèques doivent limiter *password* à une longueur raisonnable (par ex. 1024). *salt* doit être de 16 octets ou plus et provenir d'une source

correcte, par ex. `os.urandom()`.

n est le facteur de coût CPU/Mémoire, *r* la taille de bloc, *p* le facteur de parallélisation et *maxmem* limite l'utilisation de la mémoire (OpenSSL 1.1.0 limite à 32 Mio par défaut). *dklen* est la longueur de la clé dérivée en octets.

Ajouté dans la version 3.6.

15.1.9 BLAKE2

BLAKE2 est une fonction de hachage cryptographique définie dans la **RFC 7693** et disponible en deux versions :

— **BLAKE2b**, optimisée pour les plates-formes 64 bits et produisant des condensats de toutes tailles entre 1 et 64 octets,

— **BLAKE2s**, optimisée pour les plates-formes de 8 à 32 bits et produisant des empreintes de toutes tailles entre 1 et 32 octets.

BLAKE2 gère diverses fonctionnalités comme le **keyed mode** (plus rapide et plus simple que HMAC), **salted hashing**, **personalization** et le **tree hashing**.

Les objets de calcul d'empreinte de ce module suivent l'API des objets du module `hashlib` de la bibliothèque standard.

Création d'objets de calcul d'empreinte

Les nouveaux objets de calcul d'empreinte sont créés en appelant les constructeurs :

`hashlib.`**`blake2b`**(*data=b'', *, digest_size=64, key=b'', salt=b'', person=b'', fanout=1, depth=1, leaf_size=0, node_offset=0, node_depth=0, inner_size=0, last_node=False, usedforsecurity=True*)

`hashlib.`**`blake2s`**(*data=b'', *, digest_size=32, key=b'', salt=b'', person=b'', fanout=1, depth=1, leaf_size=0, node_offset=0, node_depth=0, inner_size=0, last_node=False, usedforsecurity=True*)

Ces fonctions produisent l'objet de calcul d'empreinte correspondant aux algorithmes BLAKE2b ou BLAKE2s. Elles prennent ces paramètres optionnels :

— *data* : morceau initial de données à hacher, qui doit être un *objet octets-compatible*. Il ne peut être passé que comme argument positionnel.

— *digest_size* : taille en octets de l'empreinte produite.

— *key* : clé pour les codes d'authentification de message *keyed hashing* (jusqu'à 64 octets pour BLAKE2b, jusqu'à 32 octets pour BLAKE2s).

— *salt* : sel pour le hachage randomisé *randomized hashing* (jusqu'à 16 octets pour BLAKE2b, jusqu'à 8 octets pour BLAKE2s).

— *person* : chaîne de personnalisation (jusqu'à 16 octets pour BLAKE2b, jusqu'à 8 octets pour BLAKE2s).

Le tableau suivant présente les limites des paramètres généraux (en octets) :

Hash	digest_size	len(key)	len(salt)	len(person)
BLAKE2b	64	64	16	16
BLAKE2s	32	32	8	8

Note

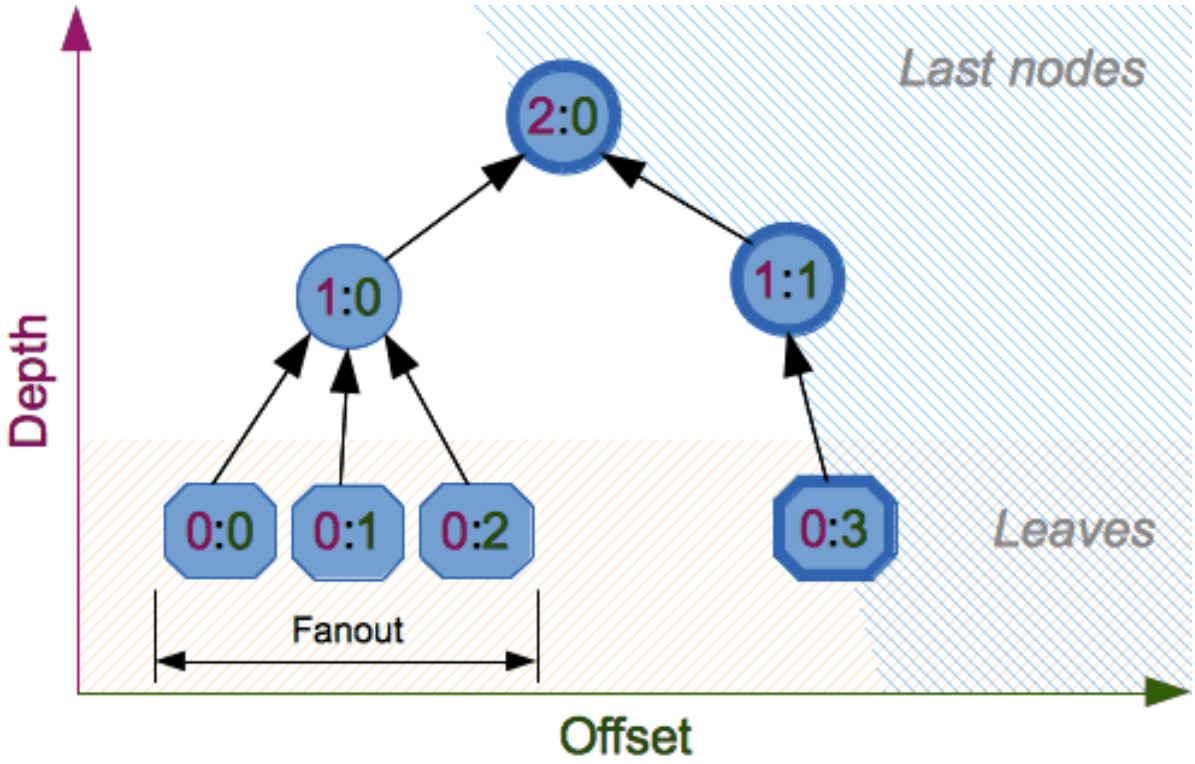
les spécifications de BLAKE2 définissent des longueurs constantes pour les sel et chaînes de personnalisation. Toutefois, par commodité, cette implémentation accepte des chaînes d'octets de n'importe quelle taille jusqu'à la longueur spécifiée. Si la longueur du paramètre est moindre par rapport à celle spécifiée, il est complété par des zéros. Ainsi, par exemple, `b'salt'` et `b'salt\x00'` sont la même valeur (ce n'est pas le cas pour *key*).

Ces tailles sont disponibles en tant que *constantes* du module et décrites ci-dessous.

Les fonctions constructeurs acceptent aussi les paramètres suivants pour le calcul d'empreintes en mode arbre :

— *fanout* : *étalement* (0 à 255, 0 si illimité, 1 correspond au mode séquentiel).

— *depth* : profondeur maximale de l'arbre (1 à 255, 255 si illimité, 1 correspond au mode séquentiel).
— *leaf_size* : taille maximale en octets d'une feuille (0 à `2**32-1`, 0 si illimité ou en mode séquentiel).
— *node_offset* : décalage du nœud (0 à `2**64-1` pour BLAKE2b, 0 à `2**48-1` pour BLAKE2s, 0 pour la première feuille la plus à gauche, ou en mode séquentiel).
— *node_depth* : profondeur du nœud (0 à 255, 0 pour les feuilles, ou en mode séquentiel).
— *inner_size* : taille de l'empreinte interne (0 à 64 pour BLAKE2b, 0 à 32 pour BLAKE2s, 0 en mode séquentiel).
— *last_node* : booléen indiquant si le nœud traité est le dernier (`False` pour le mode séquentiel).



Voir la section 2.10 dans BLAKE2 specification pour une approche globale du hachage en mode arbre.

Constantes

`blake2b.`**`SALT_SIZE`**

`blake2s.`**`SALT_SIZE`**

Longueur du sel (longueur maximale acceptée par les constructeurs).

`blake2b.`**`PERSON_SIZE`**

`blake2s.`**`PERSON_SIZE`**

Longueur de la chaîne de personnalisation (longueur maximale acceptée par les constructeurs).

`blake2b.`**`MAX_KEY_SIZE`**

`blake2s.`**`MAX_KEY_SIZE`**

Taille maximale de clé.

`blake2b.`**`MAX_DIGEST_SIZE`**

`blake2s.`**`MAX_DIGEST_SIZE`**

Taille maximale du message que peut fournir la fonction de hachage.

Exemples

Hachage simple

Pour calculer l'empreinte (ou *hash*) d'une quantité de données, vous devez d'abord construire un objet de calcul d'empreinte en appelant la fonction constructeur appropriée (`blake2b()` or `blake2s()`), ensuite le mettre à jour avec les données en appelant la méthode `update()` sur l'objet, et, pour finir, récupérer l'empreinte du message en appelant la méthode `digest()` (ou `hexdigest()` pour les chaînes hexadécimales).

```
>>> from hashlib import blake2b
>>> h = blake2b()
>>> h.update(b'Hello world')
>>> h.hexdigest()

↪'6ff843ba685842aa82031d3f53c48b66326df7639a63d128974c5c14f31a0f33343a8c65551134ed1ae0f2b0dd2bb4
↪'
```

En plus court, vous pouvez passer directement au constructeur, comme argument positionnel, le premier morceau du message pour le mettre directement à jour :

```
>>> from hashlib import blake2b
>>> blake2b(b'Hello world').hexdigest()

↪'6ff843ba685842aa82031d3f53c48b66326df7639a63d128974c5c14f31a0f33343a8c65551134ed1ae0f2b0dd2bb4
↪'
```

Vous pouvez appeler la méthode `hash.update()` autant de fois que nécessaire pour mettre à jour l'empreinte de manière itérative :

```
>>> from hashlib import blake2b
>>> items = [b'Hello', b' ', b'world']
>>> h = blake2b()
>>> for item in items:
...     h.update(item)
...
>>> h.hexdigest()

↪'6ff843ba685842aa82031d3f53c48b66326df7639a63d128974c5c14f31a0f33343a8c65551134ed1ae0f2b0dd2bb4
↪'
```

Production de tailles d'empreintes différentes

BLAKE2 permet de configurer la taille des empreintes jusqu'à 64 octets pour BLAKE2b et jusqu'à 32 octets pour BLAKE2s. Par exemple, pour remplacer SHA-1 par BLAKE2b sans changer la taille de la sortie, nous pouvons dire à BLAKE2b de produire une empreinte de 20 octets :

```
>>> from hashlib import blake2b
>>> h = blake2b(digest_size=20)
>>> h.update(b'Replacing SHA1 with the more secure function')
>>> h.hexdigest()
'd24f26cf8de66472d58d4e1b1774b4c9158b1f4c'
>>> h.digest_size
20
>>> len(h.digest())
20
```

Les objets de calcul d'empreinte initialisés avec des tailles d'empreintes différentes ont des sorties complètement différentes (les condensats courts *ne sont pas* des préfixes de condensats plus longs) ; BLAKE2b et BLAKE2s produisent des sorties différentes même si les longueurs de sortie sont les mêmes :

```
>>> from hashlib import blake2b, blake2s
>>> blake2b(digest_size=10).hexdigest()
'6fa1d8fcfd719046d762'
>>> blake2b(digest_size=11).hexdigest()
'eb6ec15daf9546254f0809'
>>> blake2s(digest_size=10).hexdigest()
'1bf21a98c78a1c376ae9'
>>> blake2s(digest_size=11).hexdigest()
'567004bf96e4a25773ebf4'
```

Hachage avec clé – Code d'authentification de message

Le hachage avec clé (*keyed hashing* en anglais) est une alternative plus simple et plus rapide à un code d'authentification de message de hachage à clé (HMAC). BLAKE2 peut être utilisé de manière sécurisée dans le mode préfixe MAC grâce à la propriété d'indifférentiabilité héritée de BLAKE.

Cet exemple montre comment obtenir un code d'authentification de message de 128 bits (en hexadécimal) pour un message `b'message data'` avec la clé `b'pseudorandom key'` :

```
>>> from hashlib import blake2b
>>> h = blake2b(key=b'pseudorandom key', digest_size=16)
>>> h.update(b'message data')
>>> h.hexdigest()
'3d363ff7401e02026f4a4687d4863ced'
```

Comme exemple pratique, une application web peut chiffrer symétriquement les *cookies* envoyés aux utilisateurs et les vérifier plus tard pour être certaine qu'ils n'ont pas été altérés :

```
>>> from hashlib import blake2b
>>> from hmac import compare_digest
>>>
>>> SECRET_KEY = b'pseudorandomly generated server secret key'
>>> AUTH_SIZE = 16
>>>
>>> def sign(cookie):
...     h = blake2b(digest_size=AUTH_SIZE, key=SECRET_KEY)
...     h.update(cookie)
...     return h.hexdigest().encode('utf-8')
>>>
>>> def verify(cookie, sig):
...     good_sig = sign(cookie)
...     return compare_digest(good_sig, sig)
>>>
>>> cookie = b'user-alice'
>>> sig = sign(cookie)
>>> print("{0},{1}".format(cookie.decode('utf-8'), sig))
user-alice,b'43b3c982cf697e0c5ab22172d1ca7421'
>>> verify(cookie, sig)
True
>>> verify(b'user-bob', sig)
False
>>> verify(cookie, b'0102030405060708090a0b0c0d0e0f00')
False
```

Même s'il possède en natif la création de code d'authentification de message (MAC), BLAKE2 peut, bien sûr, être utilisé pour construire un HMAC en combinaison avec le module `hmac` :

```
>>> import hmac, hashlib
>>> m = hmac.new(b'secret key', digestmod=hashlib.blake2s)
>>> m.update(b'message')
>>> m.hexdigest()
'e3c8102868d28b5ff85fc35dda07329970d1a01e273c37481326fe0c861c8142'
```

Hachage randomisé

En définissant le paramètre *salt* les utilisateurs peuvent introduire de l'aléatoire dans la fonction de hachage. Le hachage randomisé est utile pour se protéger des attaques par collisions sur les fonctions de hachage utilisées dans les signatures numériques.

> Le hachage aléatoire est conçu pour les situations où une partie, le préparateur du message, génère tout ou partie d'un message à signer par une seconde partie, le signataire du message. Si le préparateur du message est capable de trouver des collisions sur la fonction cryptographique de hachage (c.-à-d. deux messages produisant la même valeur une fois hachés), alors il peut préparer plusieurs versions du message, ayant un sens, qui produiront les mêmes empreintes et même signature mais avec des résultats différents (par exemple transférer 1 000 000 $ sur un compte plutôt que 10 $). Les fonctions cryptographiques de hachage ont été conçues avec comme but premier de résister aux collisions, mais la concentration actuelle d'attaques sur les fonctions de hachage peut avoir pour conséquence qu'une fonction de hachage donnée soit moins résistante qu'attendu. Le hachage aléatoire offre au signataire une protection supplémentaire en réduisant la probabilité que le préparateur puisse générer deux messages ou plus qui renverront la même empreinte lors du processus de génération de la signature — même s'il existe un moyen pratique de trouver des collisions sur la fonction de hachage. Toutefois, l'utilisation du hachage aléatoire peut réduire le niveau de sécurité fourni par une signature numérique si tous les morceaux du message sont préparés par le signataire.
>
> (NIST SP-800-106 "Randomized Hashing for Digital Signatures", article en anglais)

Dans BLAKE2, le sel est passé une seule fois lors de l'initialisation de la fonction de hachage, plutôt qu'à chaque appel de la fonction de hachage.

Avertissement

Salted hashing (ou un hachage simple) avec BLAKE2 ou toute autre fonction de hachage générique, comme SHA-256, ne convient pas pour obscurcir des mots de passe. Voir la FAQ BLAKE2 (page en anglais) pour plus d'informations.

```
>>> import os
>>> from hashlib import blake2b
>>> msg = b'some message'
>>> # Calculate the first hash with a random salt.
>>> salt1 = os.urandom(blake2b.SALT_SIZE)
>>> h1 = blake2b(salt=salt1)
>>> h1.update(msg)
>>> # Calculate the second hash with a different random salt.
>>> salt2 = os.urandom(blake2b.SALT_SIZE)
>>> h2 = blake2b(salt=salt2)
>>> h2.update(msg)
>>> # The digests are different.
>>> h1.digest() != h2.digest()
True
```

Personnalisation

Parfois il est utile de forcer une fonction de hachage à produire différentes empreintes de message d'une même entrée pour différentes utilisations. Pour citer les auteurs de la fonction de hachage Skein :

> Nous recommandons que tous les développeurs d'application considèrent sérieusement de faire cela ; nous avons vu de nombreux protocoles où une empreinte était calculée à un endroit du protocole pour être utilisée à un autre endroit car deux calculs d'empreintes étaient réalisés sur des données similaires ou liées, et qu'un attaquant peut forcer une application à prendre en entrée la même empreinte. Personnaliser chaque fonction de hachage utilisée dans le protocole stoppe immédiatement ce genre d'attaque.
>
> (The Skein Hash Function Family, p. 21, article en anglais)

BLAKE2 peut être personnalisé en passant des *bytes* à l'argument *person* :

```
>>> from hashlib import blake2b
>>> FILES_HASH_PERSON = b'MyApp Files Hash'
>>> BLOCK_HASH_PERSON = b'MyApp Block Hash'
>>> h = blake2b(digest_size=32, person=FILES_HASH_PERSON)
>>> h.update(b'the same content')
>>> h.hexdigest()
'20d9cd024d4fb086aae819a1432dd2466de12947831b75c5a30cf2676095d3b4'
>>> h = blake2b(digest_size=32, person=BLOCK_HASH_PERSON)
>>> h.update(b'the same content')
>>> h.hexdigest()
'cf68fb5761b9c44e7878bfb2c4c9aea52264a80b75005e65619778de59f383a3'
```

La personnalisation et le *keyed mode* peuvent être utilisés ensemble pour dériver différentes clés à partir d'une seule.

```
>>> from hashlib import blake2s
>>> from base64 import b64decode, b64encode
>>> orig_key = b64decode(b'Rm5EPJai72qcK3RGBpW3vPNfZy5OZothY+kHY6h21KM=')
>>> enc_key = blake2s(key=orig_key, person=b'kEncrypt').digest()
>>> mac_key = blake2s(key=orig_key, person=b'kMAC').digest()
>>> print(b64encode(enc_key).decode('utf-8'))
rbPb15S/Z9t+agffno5wuhB77VbRi6F9Iv2qIxU7WHw=
>>> print(b64encode(mac_key).decode('utf-8'))
G9GtHFE1YluXY1zWPlYk1e/nWfu0WSEb0KRcjhDeP/o=
```

Mode Arbre

L'exemple ci-dessous présente comment hacher un arbre minimal avec deux nœuds terminaux :

```
  10
 /  \
00  01
```

Cet exemple utilise en interne des empreintes de 64 octets, et produit finalement des empreintes 32 octets :

```
>>> from hashlib import blake2b
>>>
>>> FANOUT = 2
>>> DEPTH = 2
>>> LEAF_SIZE = 4096
>>> INNER_SIZE = 64
>>>
>>> buf = bytearray(6000)
>>>
>>> # Left leaf
... h00 = blake2b(buf[0:LEAF_SIZE], fanout=FANOUT, depth=DEPTH,
```

(suite sur la page suivante)

(suite de la page précédente)

```
...                 leaf_size=LEAF_SIZE, inner_size=INNER_SIZE,
...                 node_offset=0, node_depth=0, last_node=False)
>>> # Right leaf
... h01 = blake2b(buf[LEAF_SIZE:], fanout=FANOUT, depth=DEPTH,
...                 leaf_size=LEAF_SIZE, inner_size=INNER_SIZE,
...                 node_offset=1, node_depth=0, last_node=True)
>>> # Root node
... h10 = blake2b(digest_size=32, fanout=FANOUT, depth=DEPTH,
...                 leaf_size=LEAF_SIZE, inner_size=INNER_SIZE,
...                 node_offset=0, node_depth=1, last_node=True)
>>> h10.update(h00.digest())
>>> h10.update(h01.digest())
>>> h10.hexdigest()
'3ad2a9b37c6070e374c7a8c508fe20ca86b6ed54e286e93a0318e95e881db5aa'
```

Crédits

BLAKE2 a été conçu par *Jean-Philippe Aumasson*, *Samuel Neves*, *Zooko Wilcox-O'Hearn*, et *Christian Winnerlein*. Il est basé la version BLAKE qui a participé à la finale de la compétition du NIST SHA-3 créée par *Jean-Philippe Aumasson*, *Luca Henzen*, *Willi Meier*, et *Raphael C.-W. Phan*.

Il utilise le cœur de l'algorithme de chiffrement de ChaCha conçu par *Daniel J. Bernstein*.

L'implémentation dans la bibliothèque standard est basée sur le module pyblake2. Elle a été écrite par *Dmitry Chestnykh* et est basée sur l'implémentation C écrite par *Samuel Neves*. La documentation a été copiée depuis pyblake2 et écrite par *Dmitry Chestnykh*.

Le code C a été partiellement réécrit pour Python par *Christian Heimes*.

Le transfert dans le domaine public s'applique pour l'implémentation C de la fonction de hachage, ses extensions et cette documentation :

> Tout en restant dans les limites de la loi, le(s) auteur(s) a (ont) donné tous les droits d'auteur et droits connexes et voisins de ce logiciel au domaine public dans le monde entier. Ce logiciel est distribué sans aucune garantie.
>
> Vous devriez recevoir avec ce logiciel une copie de la licence *CC0 Public Domain Dedication*. Sinon, voir https://creativecommons.org/publicdomain/zero/1.0/.

Les personnes suivantes ont aidé au développement ou contribué aux modifications du projet et au domaine public selon la licence Creative Commons Public Domain Dedication 1.0 Universal :

— *Alexandr Sokolovskiy*

Voir aussi

Module `hmac`
Un module pour générer des codes d'authentification utilisant des *hash*.

Module `base64`
Un autre moyen d'encoder des *hash* binaires dans des environnements non binaires.

https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.180-4.pdf
La publication FIPS 180-4 sur les algorithmes de hachage sécurisés.

https://csrc.nist.gov/pubs/fips/202/final
La publication FIPS 202 sur le standard SHA-3.

https://www.blake2.net/
Site officiel de BLAKE2.

https://en.wikipedia.org/wiki/Cryptographic_hash_function
Article Wikipedia contenant les informations relatives aux algorithmes ayant des problèmes et leur interprétation au regard de leur utilisation.

https://www.ietf.org/rfc/rfc8018.txt
PKCS #5 : Password-Based Cryptography Specification Version 2.1

https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-132.pdf
Recommandations du NIST pour la dérivation de clé à partir de mot de passe (ressource en anglais).

15.2 `hmac` --- Keyed-Hashing for Message Authentication

Code source : Lib/hmac.py

Ce module implémente l'algorithme HMAC tel que décrit par la **RFC 2104**.

`hmac.`**`new`**(*key*, *msg=None*, *digestmod*)

Renvoie un nouvel objet HMAC. *key* est un objet *byte* ou *bytearray* représentant la clé secrète. Si *msg* est présent, un appel à `update(msg)` est effectué. *digestmod* permet de choisir l'algorithme à utiliser par l'objet HMAC, il accepte un nom de fonction de hachage (peut être tout ce qui convient à `hashlib.new()`), un constructeur de fonction de hachage ou un module implémentant la PEP 247. Bien qu'il soit après *msg*, *digestmod* est un paramètre obligatoire.

Modifié dans la version 3.4 : Le paramètre *key* peut être un *byte* ou un objet *bytearray*. Le paramètre *msg* peut être de n'importe quel type pris en charge par `hashlib`. Le paramètre *digestmod* peut être le nom d'un algorithme de hachage.

Modifié dans la version 3.8 : The *digestmod* argument is now required. Pass it as a keyword argument to avoid awkwardness when you do not have an initial *msg*.

`hmac.`**`digest`**(*key*, *msg*, *digest*)

Renvoie le code d'authentification de *msg*, pour la clé secrète *key* et à l'algorithme *digest* donné. La fonction est équivalente à `HMAC(key, msg, digest).digest()`, mais elle utilise une implémentation optimisée en C ou *inline*, qui est plus rapide pour les messages dont la taille leur permet de tenir en mémoire vive. Les paramètres *key*, *msg* et *digest* ont la même signification que pour `new()`.

Détail d'implémentation CPython, l'implémentation C optimisée n'est utilisée que lorsque le *digest* est une chaîne de caractères et le nom d'un algorithme de hachage implémenté dans OpenSSL.

Ajouté dans la version 3.7.

Un objet HMAC a les méthodes suivantes :

`HMAC.`**`update`**(*msg*)

Met à jour l'objet HMAC avec *msg*. Des appels répétés sont équivalents à un seul appel avec la concaténation de tous les arguments : `m.update(a); m.update(b)` est équivalent à `m.update(a + b)`.

Modifié dans la version 3.4 : Le paramètre *msg* peut être de n'importe quel type géré par `hashlib`.

`HMAC.`**`digest`**()

Renvoie le condensat des octets passés à la méthode `update()` jusque là. L'objet *bytes* renvoyé sera de la même longueur que la *digest_size* de la fonction de hachage donnée au constructeur. Il peut contenir des octets qui ne sont pas dans la table ASCII, y compris des octets NUL.

> **Avertissement**
>
> When comparing the output of `digest()` to an externally supplied digest during a verification routine, it is recommended to use the `compare_digest()` function instead of the == operator to reduce the vulnerability to timing attacks.

`HMAC.hexdigest()`

Comme `digest()` sauf que ce condensat est renvoyé en tant que chaîne de caractères de taille doublée contenant seulement des chiffres hexadécimaux. Cela permet d'échanger le résultat sans problèmes par e-mail ou dans d'autres environnements ne gérant pas les données binaires.

> **Avertissement**
>
> When comparing the output of `hexdigest()` to an externally supplied digest during a verification routine, it is recommended to use the `compare_digest()` function instead of the `==` operator to reduce the vulnerability to timing attacks.

`HMAC.copy()`

Renvoie une copie (un clone) de l'objet HMAC. C'est utile pour calculer de manière efficace les empreintes cryptographiques de chaînes de caractères qui ont en commun une sous-chaîne initiale.

Un objet *code d'authentification de message* (HMAC) possède les attributs suivants :

`HMAC.digest_size`

La taille du code d'authentification (c.-à-d. de l'empreinte cryptographique) en octets.

`HMAC.block_size`

La taille interne d'un bloc de l'algorithme de hachage en octets.

Ajouté dans la version 3.4.

`HMAC.name`

Le nom canonique de ce HMAC, toujours en lettres minuscules, par exemple `hmac-md5`.

Ajouté dans la version 3.4.

Modifié dans la version 3.10 : Removed the undocumented attributes `HMAC.digest_cons`, `HMAC.inner`, and `HMAC.outer`.

Ce module fournit également la fonction utilitaire suivante :

`hmac.compare_digest(a, b)`

Renvoie `a == b`. Cette fonction a été conçue pour prévenir les attaques temporelles en évitant l'implémentation de courts-circuits basés sur le contenu, ce qui la rend appropriée pour de la cryptographie. *a* et *b* doivent être du même type : soit `str` (caractères ASCII seulement, comme retourné par `HMAC.hexdigest()` par exemple), ou *bytes-like object*.

> **Note**
>
> Si *a* et *b* sont de longueurs différentes ou si une erreur se produit, une attaque temporelle pourrait en théorie obtenir des informations sur les types et longueurs de *a* et de *b*, mais pas sur leurs valeurs.

Ajouté dans la version 3.3.

Modifié dans la version 3.10 : Cette fonction utilise la fonction `CRYPTO_memcmp()` de OpenSSL quand celle-ci est disponible.

> **Voir aussi**
>
> **Module `hashlib`**
> Le module Python fournissant des fonctions de hachage sécurisé.

15.3 secrets --- Generate secure random numbers for managing secrets

Ajouté dans la version 3.6.

Code source : Lib/secrets.py

Le module *secrets* permet de générer des nombres aléatoires forts au sens de la cryptographie, adaptés à la gestion des mots de passe, à l'authentification des comptes, à la gestion des jetons de sécurité et des secrets associés.

Il faut préférer *secrets* par rapport au générateur pseudo-aléatoire du module *random*, ce dernier étant conçu pour la modélisation et la simulation, et non pour la sécurité ou la cryptographie.

> **Voir aussi**
>
> **PEP 506**

15.3.1 Nombres aléatoires

Le module *secrets* fournit un accès à la source d'aléa la plus sûre disponible sur votre système d'exploitation.

class secrets.**SystemRandom**

Classe permettant de générer des nombres aléatoires à partir des sources d'aléa les plus sûres fournies par le système d'exploitation. Se référer à *random.SystemRandom* pour plus de détails.

secrets.**choice**(*seq*)

Return a randomly chosen element from a non-empty sequence.

secrets.**randbelow**(*exclusive_upper_bound*)

Return a random int in the range [0, *exclusive_upper_bound*).

secrets.**randbits**(*k*)

Return a non-negative int with *k* random bits.

15.3.2 Génération de jetons

Le module *secrets* fournit des fonctions pour la génération sécurisée de jetons adaptés à la réinitialisation de mots de passe, à la production d'URLs difficiles à deviner, etc.

secrets.**token_bytes**([*nbytes=None*])

Renvoie une chaîne d'octets aléatoire contenant *nbytes* octets. Si *nbytes* est `None` ou omis, une valeur par défaut raisonnable est utilisée.

```
>>> token_bytes(16)
b'\xebr\x17D*t\xae\xd4\xe3S\xb6\xe2\xebP1\x8b'
```

secrets.**token_hex**([*nbytes=None*])

Renvoie une chaîne de caractères aléatoire en hexadécimal. La chaîne comporte *nbytes* octets aléatoires, chaque octet étant écrit sous la forme de deux chiffres hexadécimaux. Si *nbytes* est `None` ou omis, une valeur par défaut raisonnable est utilisée.

```
>>> token_hex(16)
'f9bf78b9a18ce6d46a0cd2b0b86df9da'
```

secrets.**token_urlsafe**([*nbytes=None*])

Renvoie une chaîne de caractères aléatoire adaptée au format URL, contenant *nbytes* octets aléatoires. Le texte est encodé en base64, chaque octet produisant en moyenne 1,3 caractères. Si *nbytes* est `None` ou omis, une valeur par défaut raisonnable est utilisée.

```
>>> token_urlsafe(16)
'Drmhze6EPcv0fN_81Bj-nA'
```

Combien d'octets mon jeton doit-il comporter ?

Afin de se prémunir des attaques par force brute, les jetons doivent être suffisamment aléatoires. Malheureusement, l'augmentation de la puissance de calcul des ordinateurs leur permet de réaliser plus de tentatives dans le même laps de temps. De ce fait, le nombre de bits recommandé pour l'aléa augmente aussi. En 2015, une longueur de 32 octets (256 bits) aléatoires est généralement considérée suffisante pour les usages typiques du module `secrets`.

Si vous souhaitez gérer la longueur des jetons par vous-même, vous pouvez spécifier la quantité d'aléa à introduire dans les jetons en passant un argument `int` aux différentes fonctions `token_*`. Cet argument indique alors le nombre d'octets aléatoires utilisés pour la création du jeton.

Sinon, si aucun argument n'est passé ou si celui-ci est `None`, les fonctions `token_*` utilisent une valeur par défaut raisonnable à la place.

> **Note**
>
> Cette valeur par défaut est susceptible de changer à n'importe quel moment, y compris lors des mises à jour de maintenance.

15.3.3 Autres fonctions

`secrets.`**`compare_digest`**(*a*, *b*)

Return `True` if strings or *bytes-like objects* *a* and *b* are equal, otherwise `False`, using a "constant-time compare" to reduce the risk of timing attacks. See `hmac.compare_digest()` for additional details.

15.3.4 Recettes et bonnes pratiques

Cette section expose les recettes et les bonnes pratiques d'utilisation de `secrets` pour gérer un niveau minimal de sécurité.

Générer un mot de passe à huit caractères alphanumériques :

```
import string
import secrets
alphabet = string.ascii_letters + string.digits
password = ''.join(secrets.choice(alphabet) for i in range(8))
```

> **Note**
>
> Applications should not **store passwords in a recoverable format**, whether plain text or encrypted. They should be salted and hashed using a cryptographically strong one-way (irreversible) hash function.

Générer un mot de passe alphanumérique à dix caractères contenant au moins un caractère en minuscule, au moins un caractère en majuscule et au moins trois chiffres :

```
import string
import secrets
alphabet = string.ascii_letters + string.digits
while True:
    password = ''.join(secrets.choice(alphabet) for i in range(10))
    if (any(c.islower() for c in password)
            and any(c.isupper() for c in password)
```

(suite sur la page suivante)

(suite de la page précédente)

```
            and sum(c.isdigit() for c in password) >= 3):
        break
```

Générer une phrase de passe dans le style xkcd :

```
import secrets
# On standard Linux systems, use a convenient dictionary file.
# Other platforms may need to provide their own word-list.
with open('/usr/share/dict/words') as f:
    words = [word.strip() for word in f]
    password = ' '.join(secrets.choice(words) for i in range(4))
```

Générer une URL temporaire difficile à deviner contenant un jeton de sécurité adapté à réinitialisation d'un mot de passe :

```
import secrets
url = 'https://example.com/reset=' + secrets.token_urlsafe()
```

CHAPITRE 16

Services génériques du système d'exploitation

Les modules documentés dans ce chapitre fournissent des interfaces vers des fonctionnalités communes à la grande majorité des systèmes d'exploitation, telles que les fichiers et l'horloge. Bien que ces interfaces soient classiquement calquées sur les interfaces Unix ou C, elles sont aussi disponibles sur la plupart des autres systèmes. En voici un aperçu :

16.1 `os` --- Miscellaneous operating system interfaces

Code source : Lib/os.py

Ce module fournit une façon portable d'utiliser les fonctionnalités dépendantes du système d'exploitation. Si vous voulez simplement lire ou écrire un fichier, voir `open()`, si vous voulez manipuler les chemins de fichiers, voir le module `os.path`, et si vous voulez lire toutes les lignes de tous les fichiers de la ligne de commande, voir le module `fileinput`. Pour la création de fichiers et de répertoires temporaires, voir le module `tempfile`, et pour la manipulation de fichiers et de répertoires de haut niveau, voir le module `shutil`.

Notes sur la disponibilité de ces fonctions :

- La conception des modules natifs Python dépendants du système d'exploitation est qu'une même fonctionnalité utilise une même interface. Par exemple, la fonction `os.stat(path)` renvoie des informations sur les statistiques de *path* dans le même format (qui est originaire de l'interface POSIX).
- Les extensions propres à un certain système d'exploitation sont également disponible par le module `os`, mais les utiliser est bien entendu une menace pour la portabilité.
- Toutes les fonctions acceptant les chemins ou noms de fichiers acceptent aussi bien des *bytes* que des *string*, et si un chemin ou un nom de fichier est renvoyé, il sera du même type.
- On VxWorks, os.popen, os.fork, os.execv and os.spawn*p* are not supported.
- On WebAssembly platforms, Android and iOS, large parts of the `os` module are not available or behave differently. APIs related to processes (e.g. `fork()`, `execve()`) and resources (e.g. `nice()`) are not available. Others like `getuid()` and `getpid()` are emulated or stubs. WebAssembly platforms also lack support for signals (e.g. `kill()`, `wait()`).

Note

Toutes les fonctions de ce module lèvent une `OSError` (ou une de ses sous-classes) dans le cas d'un chemin ou nom de fichier invalide ou inaccessible, ou si d'autres arguments sont de type correct mais non géré par le système d'exploitation.

exception os.**error**

Un alias pour les exceptions natives *OSError*.

os.**name**

Le nom du module importé dépendant du système d'exploitation. Les noms suivants ont actuellement été enregistrés : 'posix', 'nt', 'java'.

> **Voir aussi**
>
> *sys.platform* has a finer granularity. *os.uname()* gives system-dependent version information.
>
> Le module *platform* fournit des vérifications détaillées pour l'identité du système.

16.1.1 Noms de fichiers, arguments en ligne de commande, et variables d'environnement

In Python, file names, command line arguments, and environment variables are represented using the string type. On some systems, decoding these strings to and from bytes is necessary before passing them to the operating system. Python uses the *filesystem encoding and error handler* to perform this conversion (see *sys.getfilesystemencoding()*).

L'*encodage du système de fichiers et gestionnaire d'erreurs associé* sont configurés au démarrage de Python par la fonction PyConfig_Read() : regardez filesystem_encoding et filesystem_errors dans les membres de PyConfig.

Modifié dans la version 3.1 : On some systems, conversion using the file system encoding may fail. In this case, Python uses the *surrogateescape encoding error handler*, which means that undecodable bytes are replaced by a Unicode character U+DC*xx* on decoding, and these are again translated to the original byte on encoding.

The *file system encoding* must guarantee to successfully decode all bytes below 128. If the file system encoding fails to provide this guarantee, API functions can raise *UnicodeError*.

Voir aussi *encodage régional*.

16.1.2 Le mode UTF-8 de Python

Ajouté dans la version 3.7 : Voir la **PEP 540** pour plus d'informations.

Le mode UTF-8 de Python ignore *l'encodage régional* et rend obligatoire l'utilisation de l'encodage UTF-8 :

- Utiliser UTF-8 pour *l'encodage du système de fichiers*.
- *sys.getfilesystemencoding()* returns 'utf-8'.
- *locale.getpreferredencoding()* returns 'utf-8' (the *do_setlocale* argument has no effect).
- *sys.stdin*, *sys.stdout* et *sys.stderr* utilisent tous l'UTF-8 pour l'encodage du texte, avec le *gestionnaire d'erreurs* surrogateescape activé pour *sys.stdin* et *sys.stdout* (*sys.stderr* continue d'utiliser backslashreplace comme pour le mode par défaut où il prend en compte la configuration régionale).
- Sur Unix, *os.device_encoding()* renvoie 'utf-8' au lieu de l'encodage du périphérique.

Notez que la configuration des flux standards en mode UTF-8 peut être surchargée par PYTHONIOENCODING (de la même manière qu'elles peuvent l'être dans le mode par défaut qui est configuré pour la région locale).

En raison des changements apportés à ces API de bas-niveau, les API de haut-niveau ont un comportement par défaut différent :

- Les arguments en ligne de commande, les variables d'environnement et les noms de fichiers sont décodés en texte en utilisant l'UTF-8.
- *os.fsdecode()* and *os.fsencode()* use the UTF-8 encoding.
- *open()*, *io.open()*, and *codecs.open()* use the UTF-8 encoding by default. However, they still use the strict error handler by default so that attempting to open a binary file in text mode is likely to raise an exception rather than producing nonsense data.

The *Python UTF-8 Mode* is enabled if the LC_CTYPE locale is C or POSIX at Python startup (see the PyConfig_Read() function).

It can be enabled or disabled using the -X utf8 command line option and the PYTHONUTF8 environment variable.

If the `PYTHONUTF8` environment variable is not set at all, then the interpreter defaults to using the current locale settings, *unless* the current locale is identified as a legacy ASCII-based locale (as described for `PYTHONCOERCECLOCALE`), and locale coercion is either disabled or fails. In such legacy locales, the interpreter will default to enabling UTF-8 mode unless explicitly instructed not to do so.

The Python UTF-8 Mode can only be enabled at the Python startup. Its value can be read from `sys.flags.utf8_mode`.

See also the UTF-8 mode on Windows and the *filesystem encoding and error handler*.

Voir aussi

PEP 686
Le mode UTF-8 de Python sera par mis par défaut sur Python 3.15.

16.1.3 Paramètres de processus

Ces fonctions et valeurs fournissent des informations et agissent sur le processus actuel et sur l'utilisateur.

`os.ctermid()`

Renvoie l'identifiant de fichier correspondant au terminal contrôlant le processus.

Availability : Unix, not WASI.

`os.environ`

A *mapping* object where keys and values are strings that represent the process environment. For example, `environ['HOME']` is the pathname of your home directory (on some platforms), and is equivalent to `getenv("HOME")` in C.

This mapping is captured the first time the `os` module is imported, typically during Python startup as part of processing `site.py`. Changes to the environment made after this time are not reflected in `os.environ`, except for changes made by modifying `os.environ` directly.

This mapping may be used to modify the environment as well as query the environment. `putenv()` will be called automatically when the mapping is modified.

Sur Unix, les clefs et les valeurs utilisent `sys.getfilesystemencoding()` et le gestionnaire d'erreurs `surrogateescape`. Utilisez `environb` si vous désirez utiliser un encodage différent.

On Windows, the keys are converted to uppercase. This also applies when getting, setting, or deleting an item. For example, `environ['monty'] = 'python'` maps the key `'MONTY'` to the value `'python'`.

Note

Calling `putenv()` directly does not change `os.environ`, so it's better to modify `os.environ`.

Note

On some platforms, including FreeBSD and macOS, setting `environ` may cause memory leaks. Refer to the system documentation for `putenv()`.

You can delete items in this mapping to unset environment variables. `unsetenv()` will be called automatically when an item is deleted from `os.environ`, and when one of the `pop()` or `clear()` methods is called.

Voir aussi

The `os.reload_environ()` function.

Modifié dans la version 3.9 : Mis à jour pour gérer les opérateurs *merge* (|) et *update* (|=) de **PEP 584**.

os.**environb**

Bytes version of `environ` : a *mapping* object where both keys and values are `bytes` objects representing the process environment. `environ` and `environb` are synchronized (modifying `environb` updates `environ`, and vice versa).

`environb` is only available if `supports_bytes_environ` is `True`.

Ajouté dans la version 3.2.

Modifié dans la version 3.9 : Mis à jour pour gérer les opérateurs *merge* (|) et *update* (|=) de **PEP 584**.

os.**reload_environ**()

The `os.environ` and `os.environb` mappings are a cache of environment variables at the time that Python started. As such, changes to the current process environment are not reflected if made outside Python, or by `os.putenv()` or `os.unsetenv()`. Use `os.reload_environ()` to update `os.environ` and `os.environb` with any such changes to the current process environment.

> **Avertissement**
>
> This function is not thread-safe. Calling it while the environment is being modified in an other thread is an undefined behavior. Reading from `os.environ` or `os.environb`, or calling `os.getenv()` while reloading, may return an empty result.

Ajouté dans la version 3.14.

os.**chdir**(*path*)

os.**fchdir**(*fd*)

os.**getcwd**()

Ces fonctions sont décrites dans le chapitre *Fichiers et répertoires*.

os.**fsencode**(*filename*)

Encode *path-like filename* to the *filesystem encoding and error handler* ; return `bytes` unchanged.

`fsdecode()` est la fonction inverse.

Ajouté dans la version 3.2.

Modifié dans la version 3.6 : Ajout de la prise en charge des objets implémentant l'interface `os.PathLike`.

os.**fsdecode**(*filename*)

Decode the *path-like filename* from the *filesystem encoding and error handler* ; return `str` unchanged.

`fsencode()` est la fonction inverse.

Ajouté dans la version 3.2.

Modifié dans la version 3.6 : Ajout de la prise en charge des objets implémentant l'interface `os.PathLike`.

os.**fspath**(*path*)

Renvoie une représentation du chemin utilisable par le système de fichiers.

Si un objet `str` ou `bytes` est passé, il est renvoyé inchangé. Autrement, `__fspath__()` est appelée et sa valeur renvoyée tant qu'elle est un objet `str` ou `bytes`. Dans tous les autres cas, une `TypeError` est levée.

Ajouté dans la version 3.6.

class os.**PathLike**

Classe mère abstraite pour les objets représentant un chemin du système de fichiers, comme `pathlib.PurePath`.

Ajouté dans la version 3.6.

abstractmethod __fspath__()

Renvoie une représentation de l'objet utilisable par le système de fichiers.

La méthode ne devrait renvoyer que des objets `str` ou `bytes`, avec une préférence pour les `str`.

os.**getenv**(*key*, *default=None*)

Return the value of the environment variable *key* as a string if it exists, or *default* if it doesn't. *key* is a string. Note that since `getenv()` uses `os.environ`, the mapping of `getenv()` is similarly also captured on import, and the function may not reflect future environment changes.

Sur Unix, les clefs et les valeurs sont décodées avec `sys.getfilesystemencoding()` et le gestionnaire d'erreurs `surrogateescape`. Utilisez `os.getenvb()` si vous voulez utiliser un encodage différent.

Availability : Unix, Windows.

os.**getenvb**(*key*, *default=None*)

Return the value of the environment variable *key* as bytes if it exists, or *default* if it doesn't. *key* must be bytes. Note that since `getenvb()` uses `os.environb`, the mapping of `getenvb()` is similarly also captured on import, and the function may not reflect future environment changes.

`getenvb()` is only available if `supports_bytes_environ` is `True`.

Availability : Unix.

Ajouté dans la version 3.2.

os.**get_exec_path**(*env=None*)

Renvoie la liste des dossiers qui seront parcourus pour trouver un exécutable, similaire à un shell lorsque il lance un processus. *env*, quand spécifié, doit être un dictionnaire de variable d'environnement afin d'y rechercher le PATH. Par défaut quand *env* est `None`, `environ` est utilisé.

Ajouté dans la version 3.2.

os.**getegid**()

Renvoie l'identifiant du groupe effectif du processus actuel. Ça correspond au bit "*set id*" du fichier qui s'exécute dans le processus actuel.

Availability : Unix, not WASI.

os.**geteuid**()

Renvoie l'identifiant de l'utilisateur effectif du processus actuel.

Availability : Unix, not WASI.

os.**getgid**()

Renvoie l'identifiant de groupe réel du processus actuel.

Availability : Unix.

The function is a stub on WASI, see *Plateformes WebAssembly* for more information.

os.**getgrouplist**(*user*, *group*, /)

Return list of group ids that *user* belongs to. If *group* is not in the list, it is included ; typically, *group* is specified as the group ID field from the password record for *user*, because that group ID will otherwise be potentially omitted.

Availability : Unix, not WASI.

Ajouté dans la version 3.3.

os.**getgroups**()

Renvoie une liste d'identifiants de groupes additionnels associés au processus actuel.

Availability : Unix, not WASI.

> **Note**
>
> On macOS, `getgroups()` behavior differs somewhat from other Unix platforms. If the Python interpreter was built with a deployment target of `10.5` or earlier, `getgroups()` returns the list of effective group ids associated with the current user process ; this list is limited to a system-defined number of entries, typically 16, and may be modified by calls to `setgroups()` if suitably privileged. If built with a deployment target greater than `10.5`, `getgroups()` returns the current group access list for the user associated with the effective user id of the process ; the group access list may change over the lifetime of the process, it is not affected by calls to `setgroups()`, and its length is not limited to 16. The deployment target value, `MACOSX_DEPLOYMENT_TARGET`, can be obtained with `sysconfig.get_config_var()`.

os.**getlogin**()

Renvoie le nom de l'utilisateur connecté sur le terminal contrôlant le processus. Dans la plupart des cas, il est plus utile d'utiliser `getpass.getuser()` puisque cette fonction consulte les variables d'environnement `LOGNAME` et `USERNAME` pour savoir qui est l'utilisateur, et se replie finalement sur `pwd.getpwduid(os.getuid())[0]` pour avoir le nom lié à l'identifiant de l'utilisateur courant.

Availability : Unix, Windows, not WASI.

os.**getpgid**(*pid*)

Renvoie l'identifiant de groupe de processus du processus de PID *pid*. Si *pid* vaut 0, l'identifiant de groupe de processus du processus actuel est renvoyé.

Availability : Unix, not WASI.

os.**getpgrp**()

Renvoie l'identifiant du groupe de processus actuel.

Availability : Unix, not WASI.

os.**getpid**()

Renvoie l'identifiant du processus actuel.

The function is a stub on WASI, see *Plateformes WebAssembly* for more information.

os.**getppid**()

Renvoie l'identifiant du processus parent. Quand le processus parent est terminé, sur Unix, l'identifiant renvoyé est 1 pour le processus *init*, sur Windows, c'est toujours le même id, qui peut déjà avoir été réutilisé par un autre processus.

Availability : Unix, Windows, not WASI.

Modifié dans la version 3.2 : Prise en charge sur Windows.

os.**getpriority**(*which*, *who*)

Récupère la priorité d'ordonnancement des programmes. La valeur *which* est une des constantes suivantes : `PRIO_PROCESS`, `PRIO_PGRP`, ou `PRIO_USER`, et la valeur *who* est interprétée par rapport à *which* (un id de processus pour `PRIO_PROCESS`, un id de groupe de processus pour `PRIO_PGRP`, et un id d'utilisateur pour `PRIO_USER`). Une valeur nulle pour *who* dénote (respectivement) le processus appelant, le groupe de processus du processus appelant, ou l'identifiant d'utilisateur réel du processus appelant.

Availability : Unix, not WASI.

Ajouté dans la version 3.3.

os.**PRIO_PROCESS**

os.**PRIO_PGRP**

os.**PRIO_USER**

Paramètres pour les fonctions `getpriority()` et `setpriority()`.

Availability : Unix, not WASI.

Ajouté dans la version 3.3.

os.**PRIO_DARWIN_THREAD**

os.**PRIO_DARWIN_PROCESS**

os.**PRIO_DARWIN_BG**

os.**PRIO_DARWIN_NONUI**

Paramètres pour les fonctions `getpriority()` et `setpriority()`.

Availability : macOS

Ajouté dans la version 3.12.

os.**getresuid**()

Renvoie un triplet (*ruid*, *euid*, *suid*) dénotant les identifiants de l'utilisateur réel, effectif et sauvé du processus actuel.

Availability : Unix, not WASI.

Ajouté dans la version 3.2.

os.**getresgid**()

Renvoie un triplet (*rgid*, *egid*, *sgid*) dénotant les identifiants des groupes de processus réel effectif, et sauvé du processus actuel.

Availability : Unix, not WASI.

Ajouté dans la version 3.2.

os.**getuid**()

Renvoie l'identifiant réel du processus actuel.

Availability : Unix.

The function is a stub on WASI, see *Plateformes WebAssembly* for more information.

os.**initgroups**(*username*, *gid*, /)

Appelle la fonction système *initgroups* pour initialiser la liste des groupes d'accès des groupes dont *username* est membre, plus le groupe spécifié par *gid*.

Availability : Unix, not WASI, not Android.

Ajouté dans la version 3.2.

os.**putenv**(*key*, *value*, /)

Assigne la chaîne de caractères *value* à la variable d'environnement *key*. De tels changements à l'environnement affectent les sous-processus lancés par `os.system()`, `popen()` ou `fork()` et `execv()`.

Assignments to items in `os.environ` are automatically translated into corresponding calls to `putenv()`; however, calls to `putenv()` don't update `os.environ`, so it is actually preferable to assign to items of `os.environ`. This also applies to `getenv()` and `getenvb()`, which respectively use `os.environ` and `os.environb` in their implementations.

See also the `os.reload_environ()` function.

> **Note**
>
> On some platforms, including FreeBSD and macOS, setting `environ` may cause memory leaks. Refer to the system documentation for `putenv()`.

Lève un *événement d'audit* `os.putenv` avec les arguments `key`, `value`.

Modifié dans la version 3.9 : The function is now always available.

os.**setegid**(*egid*, /)

Définit l'identifiant du groupe de processus effectif du processus actuel.

Availability : Unix, not WASI, not Android.

os.**seteuid**(*euid*, /)

Définit l'identifiant de l'utilisateur effectif du processus actuel.

Availability : Unix, not WASI, not Android.

os.**setgid**(*gid*, /)

Définit l'identifiant du groupe de processus actuel.

Availability : Unix, not WASI, not Android.

os.**setgroups**(*groups*, /)

Place *groups* dans la liste d'identifiants de groupes additionnels associée. *groups* doit être une séquence, et chaque élément doit être un nombre entier identifiant un groupe. Cette opération est typiquement disponible uniquement pour le super utilisateur.

Availability : Unix, not WASI.

> **Note**
>
> On macOS, the length of *groups* may not exceed the system-defined maximum number of effective group ids, typically 16. See the documentation for `getgroups()` for cases where it may not return the same group list set by calling setgroups().

os.**setns**(*fd*, *nstype=0*)

Reassociate the current thread with a Linux namespace. See the `setns(2)` and `namespaces(7)` man pages for more details.

If *fd* refers to a `/proc/pid/ns/` link, `setns()` reassociates the calling thread with the namespace associated with that link, and *nstype* may be set to one of the *CLONE_NEW* constants* to impose constraints on the operation (`0` means no constraints).

Since Linux 5.8, *fd* may refer to a PID file descriptor obtained from `pidfd_open()`. In this case, `setns()` reassociates the calling thread into one or more of the same namespaces as the thread referred to by *fd*. This is subject to any constraints imposed by *nstype*, which is a bit mask combining one or more of the *CLONE_NEW* constants*, e.g. `setns(fd, os.CLONE_NEWUTS | os.CLONE_NEWPID)`. The caller's memberships in unspecified namespaces are left unchanged.

fd can be any object with a `fileno()` method, or a raw file descriptor.

This example reassociates the thread with the `init` process's network namespace :

```
fd = os.open("/proc/1/ns/net", os.O_RDONLY)
os.setns(fd, os.CLONE_NEWNET)
os.close(fd)
```

Availability : Linux >= 3.0 with glibc >= 2.14.

Ajouté dans la version 3.12.

> **Voir aussi**
>
> The `unshare()` function.

os.**setpgrp**()

Call the system call `setpgrp()` or `setpgrp(0, 0)` depending on which version is implemented (if any). See the Unix manual for the semantics.

Availability : Unix, not WASI.

os.**setpgid**(*pid*, *pgrp*, /)

Call the system call `setpgid()` to set the process group id of the process with id *pid* to the process group

with id *pgrp*. See the Unix manual for the semantics.

Availability : Unix, not WASI.

os.**setpriority**(*which*, *who*, *priority*)

Définit la priorité d'ordonnancement des programmes. La valeur *which* est une des constantes suivantes : `PRIO_PROCESS`, `PRIO_PGRP`, ou `PRIO_USER`, et *who* est interprété en fonction de *which* (un PID pour `PRIO_PROCESS`, un identifiant de groupe de processus pour `PRIO_PGRP`, et un identifiant d'utilisateur pour `PRIO_USER`). Une valeur nulle pour *who* dénote (respectivement) le processus appelant, le groupe de processus du processus appelant, ou l'identifiant de l'utilisateur réel du processus appelant. *priority* est une valeur comprise entre `-20` et `19`. La priorité par défaut est 0 ; les priorités plus faibles amènent à un ordonnancement plus favorable.

Availability : Unix, not WASI.

Ajouté dans la version 3.3.

os.**setregid**(*rgid*, *egid*, /)

Définit l'identifiant des groupes réel et effectif du processus actuel.

Availability : Unix, not WASI, not Android.

os.**setresgid**(*rgid*, *egid*, *sgid*, /)

Définit l'identifiant des groupes réel, effectif et sauvé du processus actuel.

Availability : Unix, not WASI, not Android.

Ajouté dans la version 3.2.

os.**setresuid**(*ruid*, *euid*, *suid*, /)

Définit l'identifiant des utilisateurs réel, effectif et sauvé du processus actuel.

Availability : Unix, not WASI, not Android.

Ajouté dans la version 3.2.

os.**setreuid**(*ruid*, *euid*, /)

Définit l'identifiant des utilisateurs réel et effectif du processus actuel.

Availability : Unix, not WASI, not Android.

os.**getsid**(*pid*, /)

Call the system call `getsid()`. See the Unix manual for the semantics.

Availability : Unix, not WASI.

os.**setsid**()

Call the system call `setsid()`. See the Unix manual for the semantics.

Availability : Unix, not WASI.

os.**setuid**(*uid*, /)

Définit l'identifiant de l'utilisateur du processus actuel.

Availability : Unix, not WASI, not Android.

os.**strerror**(*code*, /)

Return the error message corresponding to the error code in *code*. On platforms where `strerror()` returns `NULL` when given an unknown error number, `ValueError` is raised.

os.**supports_bytes_environ**

`True` si le type natif de l'environnement du système d'exploitation est *bytes* (par exemple : `False` sur Windows).

Ajouté dans la version 3.2.

os.**umask**(*mask*, /)

Définit le *umask* actuel et renvoie la valeur précédente.

The function is a stub on WASI, see *Plateformes WebAssembly* for more information.

os.**uname**()

Renvoie des informations identifiant le système d'exploitation actuel. La valeur de retour est un objet à cinq attributs :
— sysname — nom du système d'exploitation
— nodename — nom de la machine sur le réseau (dépendant de l'implémentation)
— release — *release* du système d'exploitation
— version — version du système d'exploitation
— machine — identifiant du matériel
Pour la rétrocompatibilité, cet objet est également itérable, se comportant comme un quintuplet contenant sysname, nodename, release, version, et machine dans cet ordre.

Certains systèmes tronquent nodename à 8 caractères ou à la composante dominante. Un meilleur moyen de récupérer le nom de l'hôte est *socket.gethostname()* ou encore socket.gethostbyaddr(socket.gethostname()).

On macOS, iOS and Android, this returns the *kernel* name and version (i.e., 'Darwin' on macOS and iOS ; 'Linux' on Android). *platform.uname()* can be used to get the user-facing operating system name and version on iOS and Android.

Availability : Unix.

Modifié dans la version 3.3 : Type de retour changé d'un quintuplet en un objet compatible avec le type *n*-uplet, avec des attributs nommés.

os.**unsetenv**(*key*, /)

Supprime la variable d'environnement appelée *key*. De tels changements à l'environnement affectent les sous-processus lancés avec *os.system()*, *popen()* ou *fork()* et *execv()*.

Deletion of items in *os.environ* is automatically translated into a corresponding call to *unsetenv()* ; however, calls to *unsetenv()* don't update *os.environ*, so it is actually preferable to delete items of *os.environ*.

See also the *os.reload_environ()* function.

Lève un *événement d'audit* os.unsetenv avec l'argument key.

Modifié dans la version 3.9 : The function is now always available and is also available on Windows.

os.**unshare**(*flags*)

Disassociate parts of the process execution context, and move them into a newly created namespace. See the *unshare(2)* man page for more details. The *flags* argument is a bit mask, combining zero or more of the *CLONE_* constants*, that specifies which parts of the execution context should be unshared from their existing associations and moved to a new namespace. If the *flags* argument is 0, no changes are made to the calling process's execution context.

Availability : Linux >= 2.6.16.

Ajouté dans la version 3.12.

Voir aussi

The *setns()* function.

Flags to the *unshare()* function, if the implementation supports them. See *unshare(2)* in the Linux manual for their exact effect and availability.

os.**CLONE_FILES**

os.**CLONE_FS**

os.**CLONE_NEWCGROUP**

os.**CLONE_NEWIPC**

os.**CLONE_NEWNET**

os.**CLONE_NEWNS**

os.**CLONE_NEWPID**

os.**CLONE_NEWTIME**

os.**CLONE_NEWUSER**

os.**CLONE_NEWUTS**

os.**CLONE_SIGHAND**

os.**CLONE_SYSVSEM**

os.**CLONE_THREAD**

os.**CLONE_VM**

16.1.4 Création de fichiers objets

These functions create new *file objects*. (See also `open()` for opening file descriptors.)

os.**fdopen**(*fd*, **args*, ***kwargs*)

Renvoie un fichier objet ouvert connecté au descripteur de fichier *fd*. C'est un alias de la primitive `open()` et accepte les mêmes arguments. La seule différence est que le premier argument de `fdopen()` doit toujours être un entier.

16.1.5 Opérations sur les descripteurs de fichiers

Ces fonctions opèrent sur des flux d'entrées/sorties référencés par des descripteurs de fichiers.

Les descripteurs de fichiers sont de petits entiers correspondant à un fichier qui a été ouvert par le processus courant. Par exemple, l'entrée standard est habituellement le descripteur de fichier 0, la sortie standard est 1, et le flux standard d'erreur est 2. Les autres fichiers ouverts par un processus vont se voir assigner 3, 4, 5, etc. Le nom « descripteur de fichier » est légèrement trompeur : sur les plate-formes Unix, les connecteurs (*socket* en anglais) et les tubes (*pipe* en anglais) sont également référencés par des descripteurs.

La méthode `fileno()` peut être utilisée pour obtenir le descripteur de fichier associé à un *file object* quand nécessaire. Notez qu'utiliser le descripteur directement outrepasse les méthodes du fichier objet, ignorant donc des aspects tels que la mise en tampon interne des données.

os.**close**(*fd*)

Ferme le descripteur de fichier *fd*.

> **Note**
>
> Cette fonction est destinée aux opérations d'entrées/sorties de bas niveau et doit être appliquée à un descripteur de fichier comme ceux donnés par `os.open()` ou `pipe()`. Pour fermer un « fichier objet » renvoyé par la primitive `open()`, `popen()` ou `fdopen()`, il faut utiliser sa méthode `close()`.

os.**closerange**(*fd_low*, *fd_high*, /)

Ferme tous les descripteurs de fichiers entre *fd_low* (inclus) jusque *fd_high* (exclus), en ignorant les erreurs. Équivalent (mais beaucoup plus rapide) à :

```
for fd in range(fd_low, fd_high):
    try:
        os.close(fd)
    except OSError:
        pass
```

os.**copy_file_range**(*src*, *dst*, *count*, *offset_src=None*, *offset_dst=None*)

Copy *count* bytes from file descriptor *src*, starting from offset *offset_src*, to file descriptor *dst*, starting from offset *offset_dst*. If *offset_src* is `None`, then *src* is read from the current position ; respectively for *offset_dst*.

In Linux kernel older than 5.3, the files pointed to by *src* and *dst* must reside in the same filesystem, otherwise an `OSError` is raised with `errno` set to `errno.EXDEV`.

This copy is done without the additional cost of transferring data from the kernel to user space and then back into the kernel. Additionally, some filesystems could implement extra optimizations, such as the use of reflinks (i.e., two or more inodes that share pointers to the same copy-on-write disk blocks; supported file systems include btrfs and XFS) and server-side copy (in the case of NFS).

The function copies bytes between two file descriptors. Text options, like the encoding and the line ending, are ignored.

La valeur de retour est le nombre d'octets copiés. Cela peut être moins que le nombre demandé.

> **Note**
>
> On Linux, `os.copy_file_range()` should not be used for copying a range of a pseudo file from a special filesystem like procfs and sysfs. It will always copy no bytes and return 0 as if the file was empty because of a known Linux kernel issue.

Availability : Linux >= 4.5 with glibc >= 2.27.

Ajouté dans la version 3.8.

`os.`**`device_encoding`**(*fd*)

Renvoie une chaîne de caractères décrivant l'encodage du périphérique associé à *fd* s'il est connecté à un terminal, sinon renvoie `None`.

On Unix, if the *Python UTF-8 Mode* is enabled, return `'UTF-8'` rather than the device encoding.

Modifié dans la version 3.10 : On Unix, the function now implements the Python UTF-8 Mode.

`os.`**`dup`**(*fd*, `/`)

Renvoie une copie du descripteur de fichier *fd*. Le nouveau descripteur de fichier est *non-héritable*.

Sur Windows, quand un flux standard est dupliqué (0 : *stdin*, 1 : *stdout*, 2 : *stderr*), le nouveau descripteur de fichier est *héritable*.

Availability : not WASI.

Modifié dans la version 3.4 : Le nouveau descripteur de fichier est maintenant non-héritable.

`os.`**`dup2`**(*fd*, *fd2*, *inheritable=True*)

Copie le descripteur de fichier *fd* dans *fd2*, en fermant le second d'abord si nécessaire. Renvoie *fd2*. Le nouveau descripteur de fichier est *héritable* par défaut, ou non-héritable si *inheritable* vaut `False`.

Availability : not WASI.

Modifié dans la version 3.4 : Ajout du paramètre optionnel *inheritable*.

Modifié dans la version 3.7 : Renvoie *fd2* en cas de succès. Auparavant, `None` était toujours renvoyé.

`os.`**`fchmod`**(*fd*, *mode*)

Change le mode du fichier donné par *fd* en la valeur numérique *mode*. Voir la documentation de `chmod()` pour les valeurs possibles de *mode*. Depuis Python 3.3, c'est équivalent à `os.chmod(fd, mode)`.

Lève un *événement d'audit* `os.chmod` avec les arguments `path`, `mode`, `dir_fd`.

Availability : Unix, Windows.

The function is limited on WASI, see *Plateformes WebAssembly* for more information.

Modifié dans la version 3.13 : Added support on Windows.

`os.`**`fchown`**(*fd*, *uid*, *gid*)

Change le propriétaire et l'identifiant de groupe du fichier donné par *fd* en les valeurs numériques *uid* et *gid*. Pour laisser l'un des identifiants inchangés, mettez-le à `-1`. Voir `chown()`. Depuis Python 3.3, c'est équivalent à `os.chown(fd, uid, gid)`.

Lève un *événement d'audit* `os.chown` avec les arguments `path`, `uid`, `gid`, `dir_fd`.

Availability : Unix.

The function is limited on WASI, see *Plateformes WebAssembly* for more information.

os.**fdatasync**(*fd*)

Force l'écriture du fichier ayant le descripteur *fd* sur le disque. Ne force pas la mise à jour des méta-données.

Availability : Unix.

> **Note**
>
> Cette fonction n'est pas disponible sur MacOS.

os.**fpathconf**(*fd*, *name*, /)

Renvoie les informations de la configuration du système pour un fichier ouvert. *name* spécifie la valeur de la configuration à récupérer, ça peut être une chaîne de caractères avec le nom d'une valeur système définie ; ces valeurs sont spécifiées dans certains standards (POSIX.1, Unix 95, Unix 98, et d'autres). Certaines plate-formes définissent des noms additionnels également. Les noms connus par le système d'exploitation sont donnés dans le dictionnaire `pathconf_names`. Pour les variables de configuration qui ne sont pas inclues dans ce *mapping*, passer un entier pour *name* est également accepté.

Si *name* est une chaîne de caractères et n'est pas connu, une `ValueError` est levée. Si une valeur spécifique de *name* n'est pas gérée par le système hôte, même si elle est incluse dans `pathconf_names`, une `OSError` est levée avec `errno.EINVAL` pour code d'erreur.

Depuis Python 3.3, c'est équivalent à `os.pathconf(fd, name)`.

Availability : Unix.

os.**fstat**(*fd*)

Récupère le statut du descripteur de fichier *fd*. Renvoie un objet `stat_result`.

Depuis Python 3.3, c'est équivalent à `os.stat(fd)`.

> **Voir aussi**
>
> La fonction `stat()`.

os.**fstatvfs**(*fd*, /)

Renvoie des informations sur le système de fichier contenant le fichier associé au descripteur *fd*, comme `statvfs()`. Depuis Python 3.3, c'est équivalent à `os.statvfs(fd)`.

Availability : Unix.

os.**fsync**(*fd*)

Force write of file with filedescriptor *fd* to disk. On Unix, this calls the native `fsync()` function ; on Windows, the MS `_commit()` function.

Si vous débutez avec un *file object* Python mis en tampon *f*, commencez par faire `f.flush()` et seulement ensuite `os.fsync(f.fileno())` pour être sûr que tous les tampons internes associés à *f* soient écrits sur le disque.

Availability : Unix, Windows.

os.**ftruncate**(*fd*, *length*, /)

Tronque le fichier correspondant au descripteur *fd* pour qu'il soit maximum long de *length bytes*. Depuis Python 3.3, c'est équivalent à `os.truncate(fd, length)`.

Lève un *événement d'audit* `os.truncate` avec les arguments `fd`, `length`.

Availability : Unix, Windows.

Modifié dans la version 3.5 : Prise en charge de Windows

os.**get_blocking**(*fd*, /)

Récupère le mode bloquant du descripteur de fichier : `False` si l'indicateur `O_NONBLOCK` est mis, et `True` si l'indicateur est effacé.

Voir également `set_blocking()` et `socket.socket.setblocking()`.

Availability : Unix, Windows.

The function is limited on WASI, see *Plateformes WebAssembly* for more information.

On Windows, this function is limited to pipes.

Ajouté dans la version 3.5.

Modifié dans la version 3.12 : Added support for pipes on Windows.

os.**grantpt**(*fd*, /)

Grant access to the slave pseudo-terminal device associated with the master pseudo-terminal device to which the file descriptor *fd* refers. The file descriptor *fd* is not closed upon failure.

Calls the C standard library function `grantpt()`.

Availability : Unix, not WASI.

Ajouté dans la version 3.13.

os.**isatty**(*fd*, /)

Renvoie `True` si le descripteur de fichier *fd* est ouvert et connecté à un périphérique TTY (ou compatible), sinon `False`.

os.**lockf**(*fd*, *cmd*, *len*, /)

Applique, teste, ou retire un verrou POSIX sur un descripteur de fichier ouvert. *fd* est un descripteur de fichier ouvert. *cmd* spécifie la commande à utiliser (une des valeurs suivantes : `F_LOCK`, `F_TLOCK`, `F_ULOCK`, ou `F_TEST`). *len* spécifie la section du fichier à verrouiller.

Lève un *événement d'audit* `os.lockf` avec les arguments `fd`, `cmd`, `len`.

Availability : Unix.

Ajouté dans la version 3.3.

os.**F_LOCK**

os.**F_TLOCK**

os.**F_ULOCK**

os.**F_TEST**

Indicateurs spécifiant quelle action `lockf()` va prendre.

Availability : Unix.

Ajouté dans la version 3.3.

os.**login_tty**(*fd*, /)

Prepare the tty of which fd is a file descriptor for a new login session. Make the calling process a session leader ; make the tty the controlling tty, the stdin, the stdout, and the stderr of the calling process ; close fd.

Availability : Unix, not WASI.

Ajouté dans la version 3.11.

os.**lseek**(*fd*, *pos*, *whence*, /)

Set the current position of file descriptor *fd* to position *pos*, modified by *whence*, and return the new position in bytes relative to the start of the file. Valid values for *whence* are :

— `SEEK_SET` or `0` -- set *pos* relative to the beginning of the file
— `SEEK_CUR` or `1` -- set *pos* relative to the current file position
— `SEEK_END` or `2` -- set *pos* relative to the end of the file
— `SEEK_HOLE` -- set *pos* to the next data location, relative to *pos*
— `SEEK_DATA` -- set *pos* to the next data hole, relative to *pos*

Modifié dans la version 3.3 : Add support for `SEEK_HOLE` and `SEEK_DATA`.

os.**SEEK_SET**

os.**SEEK_CUR**

os.**SEEK_END**

Parameters to the `lseek()` function and the `seek()` method on *file-like objects*, for whence to adjust the file position indicator.

SEEK_SET
Adjust the file position relative to the beginning of the file.

SEEK_CUR
Adjust the file position relative to the current file position.

SEEK_END
Adjust the file position relative to the end of the file.

Their values are 0, 1, and 2, respectively.

os.**SEEK_HOLE**

os.**SEEK_DATA**

Parameters to the `lseek()` function and the `seek()` method on *file-like objects*, for seeking file data and holes on sparsely allocated files.

SEEK_DATA
Adjust the file offset to the next location containing data, relative to the seek position.

SEEK_HOLE
Adjust the file offset to the next location containing a hole, relative to the seek position. A hole is defined as a sequence of zeros.

> **Note**
>
> These operations only make sense for filesystems that support them.

Availability : Linux >= 3.1, macOS, Unix

Ajouté dans la version 3.3.

os.**open**(*path*, *flags*, *mode=0o777*, **, dir_fd=None*)

Ouvre le fichier *path* et met certains indicateurs selon *flags* et éventuellement son mode selon *mode*. Lors de l'évaluation de *code*, ce *umask* actuel est d'abord masquée. Renvoie le descripteur de fichier du fichier nouvellement ouvert. Le nouveau descripteur de fichier est *non-héritable*.

Pour une description des indicateurs et des valeurs des modes, voir la documentation de la bibliothèque standard du C. Les constantes d'indicateurs (telles que `O_RDONLY` et `O_WRONLY`) sont définies dans le module `os`. En particulier, sur Windows, ajouter `O_BINARY` est nécessaire pour ouvrir des fichiers en binaire.

Cette fonction prend en charge des *chemins relatifs à des descripteurs de répertoires* avec le paramètre *dir_fd*.

Lève un *événement d'audit* `open` avec les arguments `path`, `mode`, `flags`.

Modifié dans la version 3.4 : Le nouveau descripteur de fichier est maintenant non-héritable.

> **Note**
>
> Cette fonction est destinée aux E/S de bas-niveau. Pour un usage normal, utilisez la primitive `open()` qui renvoie un *file object* avec les méthodes `read()` et `write()` (et plein d'autres). Pour envelopper un descripteur de fichier dans un fichier objet, utilisez `fdopen()`.

Modifié dans la version 3.3 : Paramètre *dir_fd* ajouté.

Modifié dans la version 3.5 : Si l'appel système est interrompu et que le gestionnaire de signal ne lève aucune exception, la fonction réessaye l'appel système au lieu de lever une `InterruptedError` (voir la **PEP 475** à propos du raisonnement).

Modifié dans la version 3.6 : Accepte un *path-like object*.

Les constantes suivantes sont optionnelles pour le paramètre *flag* à la fonction `open()`. Elles peuvent être combinées en utilisant l'opérateur bit-à-bit OR `|`. certains ne sont pas disponibles sur toutes les plate-formes. Pour une description sur leur disponibilité et leur usage, consultez la page de manuel Unix `open(2)` ou la MSDN sur Windows.

os.**O_RDONLY**

os.**O_WRONLY**

os.**O_RDWR**

os.**O_APPEND**

os.**O_CREAT**

os.**O_EXCL**

os.**O_TRUNC**

Les constantes ci-dessus sont disponibles sur Unix et Windows.

os.**O_DSYNC**

os.**O_RSYNC**

os.**O_SYNC**

os.**O_NDELAY**

os.**O_NONBLOCK**

os.**O_NOCTTY**

os.**O_CLOEXEC**

Les constantes ci-dessus sont uniquement disponibles sur Unix.

Modifié dans la version 3.3 : Ajout de la constante `O_CLOCEXEC`.

os.**O_BINARY**

os.**O_NOINHERIT**

os.**O_SHORT_LIVED**

os.**O_TEMPORARY**

os.**O_RANDOM**

os.**O_SEQUENTIAL**

os.**O_TEXT**

Les constantes ci-dessus sont uniquement disponibles sur Windows.

os.**O_EVTONLY**

os.**O_FSYNC**

os.**O_SYMLINK**

os.**O_NOFOLLOW_ANY**

The above constants are only available on macOS.

Modifié dans la version 3.10 : Add `O_EVTONLY`, `O_FSYNC`, `O_SYMLINK` and `O_NOFOLLOW_ANY` constants.

os.**O_ASYNC**

os.**O_DIRECT**

os.**O_DIRECTORY**

os.**O_NOFOLLOW**

os.**O_NOATIME**

os.**O_PATH**

os.**O_TMPFILE**

os.**O_SHLOCK**

os.**O_EXLOCK**

Les constantes ci-dessus sont des extensions et ne sont pas présentes si elles ne sont pas définies par la bibliothèque C.

Modifié dans la version 3.4 : Ajout de `O_PATH` sur les systèmes qui le gèrent. Ajout de `O_TMPFILE`, uniquement disponible sur Linux Kernel 3.11 ou plus récent.

os.**openpty**()

Ouvre une nouvelle paire pseudo-terminal. Renvoie une paire de descripteurs de fichiers `(master, slave)` pour le PTY et le TTY respectivement. Les nouveaux descripteurs de fichiers sont *non-héritables*. Pour une approche (légèrement) plus portable, utilisez le module `pty`.

Availability : Unix, not WASI.

Modifié dans la version 3.4 : Les nouveaux descripteurs de fichiers sont maintenant non-héritables.

os.**pipe**()

Crée un *pipe* (un tube). Renvoie une paire de descripteurs de fichiers `(r, w)` utilisables respectivement pour lire et pour écrire. Les nouveaux descripteurs de fichiers sont *non-héritables*.

Availability : Unix, Windows.

Modifié dans la version 3.4 : Les nouveaux descripteurs de fichiers sont maintenant non-héritables.

os.**pipe2**(*flags*, /)

Crée un *pipe* avec *flags* mis atomiquement. *flags* peut être construit en appliquant l'opérateur `|` (OU) sur une ou plus de ces valeurs : `O_NONBLOCK`, `O_CLOEXEC`. Renvoie une paire de descripteurs de fichiers `(r, w)` utilisables respectivement pour lire et pour écrire.

Availability : Unix, not WASI.

Ajouté dans la version 3.3.

os.**posix_fallocate**(*fd*, *offset*, *len*, /)

Assure que suffisamment d'espace sur le disque est alloué pour le fichier spécifié par *fd* partant de *offset* et continuant sur *len* bytes.

Availability : Unix.

Ajouté dans la version 3.3.

os.**posix_fadvise**(*fd*, *offset*, *len*, *advice*, /)

Annonce une intention d'accéder à des données selon un motif spécifique, et donc permettant au noyau de faire des optimisations. Le conseil *advice* s'applique à la région spécifiée par *fd*, démarrant à *offset* et continuant sur *len* bytes. *advice* est une des valeurs suivantes : `POSIX_FADV_NORMAL`, `POSIX_FADV_SEQUENTIAL`, `POSIX_FADV_RANDOM`, `POSIX_FADV_NOREUSE`, `POSIX_FADV_WILLNEED`, ou `POSIX_FADV_DONTNEED`.

Availability : Unix.

Ajouté dans la version 3.3.

os.**POSIX_FADV_NORMAL**

os.**POSIX_FADV_SEQUENTIAL**

os.**POSIX_FADV_RANDOM**

os.**POSIX_FADV_NOREUSE**

os.**POSIX_FADV_WILLNEED**

os.**POSIX_FADV_DONTNEED**

Indicateurs qui peuvent être utilisés dans *advice* dans la fonction `posix_fadvise()` et qui spécifient le motif d'accès qui est censé être utilisé.

Availability : Unix.

Ajouté dans la version 3.3.

os.**pread**(*fd*, *n*, *offset*, /)

Lit au maximum *n* octets depuis le descripteur de fichier *fd* à la position *offset* sans modifier cette position.

Renvoie une chaîne d'octets contenant les octets lus, ou une chaîne d'octets vide si la fin du fichier pointé par *fd* est atteinte.

Availability : Unix.

Ajouté dans la version 3.3.

os.**posix_openpt**(*oflag*, /)

Open and return a file descriptor for a master pseudo-terminal device.

Calls the C standard library function `posix_openpt()`. The *oflag* argument is used to set file status flags and file access modes as specified in the manual page of `posix_openpt()` of your system.

The returned file descriptor is *non-inheritable*. If the value `O_CLOEXEC` is available on the system, it is added to *oflag*.

Availability : Unix, not WASI.

Ajouté dans la version 3.13.

os.**preadv**(*fd*, *buffers*, *offset*, *flags=0*, /)

Lit depuis un descripteur de fichier *fd*, à la position *offset* dans des *objets bytes-compatibles* mutables *buffers*, sans modifier la position dans le fichier. Les données sont transférées dans chaque tampon, jusqu'à ce qu'il soit plein, tour à tour.

L'argument *flags* contient un OU logique bit-à-bit de zéro ou plusieurs des indicateurs suivants :

— `RWF_HIPRI`

— `RWF_NOWAIT`

Renvoie le nombre total d'octets réellement lus, qui peut être inférieur à la capacité totale de tous les objets.

Le système d'exploitation peut définir une limite (valeur `sysconf()` `'SC_IOV_MAX'`) sur le nombre de mémoires tampons pouvant être utilisées.

Combine les fonctionnalités de `os.readv()` et `os.pread()`.

Availability : Linux >= 2.6.30, FreeBSD >= 6.0, OpenBSD >= 2.7, AIX >= 7.1.

Using flags requires Linux >= 4.6.

Ajouté dans la version 3.7.

os.**RWF_NOWAIT**

Ne pas attendre pour des données qui ne sont pas immédiatement disponibles. Si cette option est spécifiée, l'appel système retourne instantanément s'il doit lire les données du stockage sous-jacent ou attendre un verrou.

If some data was successfully read, it will return the number of bytes read. If no bytes were read, it will return `-1` and set errno to `errno.EAGAIN`.

Availability : Linux >= 4.14.

Ajouté dans la version 3.7.

os.**RWF_HIPRI**

Lecture/écriture haute priorité. Permet aux systèmes de fichiers de type bloc d'utiliser le *polling* du périphérique, qui fournit une latence inférieure, mais peut utiliser des ressources supplémentaires.

Actuellement, sous Linux, cette fonctionnalité est utilisable uniquement sur un descripteur de fichier ouvert à l'aide de l'option `O_DIRECT`.

Availability : Linux >= 4.6.

Ajouté dans la version 3.7.

os.**ptsname**(*fd*, /)

Return the name of the slave pseudo-terminal device associated with the master pseudo-terminal device to which the file descriptor *fd* refers. The file descriptor *fd* is not closed upon failure.

Calls the reentrant C standard library function `ptsname_r()` if it is available ; otherwise, the C standard library function `ptsname()`, which is not guaranteed to be thread-safe, is called.

Availability : Unix, not WASI.

Ajouté dans la version 3.13.

`os.`**`pwrite`**(*fd*, *str*, *offset*, /)

Écrit la chaîne d'octets de *str* dans le descripteur de fichier *fd* à la position *offset* en laissant la position dans le fichier inchangée.

Renvoie le nombre d'octets effectivement écrits.

Availability : Unix.

Ajouté dans la version 3.3.

`os.`**`pwritev`**(*fd*, *buffers*, *offset*, *flags=0*, /)

Write the *buffers* contents to file descriptor *fd* at an offset *offset*, leaving the file offset unchanged. *buffers* must be a sequence of *bytes-like objects*. Buffers are processed in array order. Entire contents of the first buffer is written before proceeding to the second, and so on.

L'argument *flags* contient un OU logique bit-à-bit de zéro ou plusieurs des indicateurs suivants :

— *`RWF_DSYNC`*
— *`RWF_SYNC`*
— *`RWF_APPEND`*

Renvoie le nombre total d'octets effectivement écrits.

Le système d'exploitation peut définir une limite (valeur *`sysconf()`* `'SC_IOV_MAX'`) sur le nombre de mémoires tampons pouvant être utilisées.

Combine les fonctionnalités de `os. writev()` et `os. pwrite()`.

Availability : Linux >= 2.6.30, FreeBSD >= 6.0, OpenBSD >= 2.7, AIX >= 7.1.

Using flags requires Linux >= 4.6.

Ajouté dans la version 3.7.

`os.`**`RWF_DSYNC`**

Provide a per-write equivalent of the *`O_DSYNC`* *`os.open()`* flag. This flag effect applies only to the data range written by the system call.

Availability : Linux >= 4.7.

Ajouté dans la version 3.7.

`os.`**`RWF_SYNC`**

Provide a per-write equivalent of the *`O_SYNC`* *`os.open()`* flag. This flag effect applies only to the data range written by the system call.

Availability : Linux >= 4.7.

Ajouté dans la version 3.7.

`os.`**`RWF_APPEND`**

Provide a per-write equivalent of the *`O_APPEND`* *`os.open()`* flag. This flag is meaningful only for *`os.pwritev()`*, and its effect applies only to the data range written by the system call. The *offset* argument does not affect the write operation ; the data is always appended to the end of the file. However, if the *offset* argument is `-1`, the current file *offset* is updated.

Availability : Linux >= 4.16.

Ajouté dans la version 3.10.

`os.`**`read`**(*fd*, *n*, /)

Lit au maximum *n* octets du descripteur de fichier *fd*.

Renvoie une chaîne d'octets contenant les octets lus, ou une chaîne d'octets vide si la fin du fichier pointé par *fd* est atteinte.

> **Note**
>
> Cette fonction est destinée aux E/S bas niveau et doit être appliquée à un descripteur de fichier comme renvoyé par `os.open()` ou `pipe()`. Pour lire dans un « fichier objet » renvoyé par la primitive `open()`, `popen()` ou `fdopen()`, ou par `stdin`, utilisez sa méthode `read()` ou `readline()`.

Modifié dans la version 3.5 : Si l'appel système est interrompu et que le gestionnaire de signal ne lève aucune exception, la fonction réessaye l'appel système au lieu de lever une `InterruptedError` (voir la **PEP 475** à propos du raisonnement).

`os.readinto`(*fd*, *buffer*, /)

Read from a file descriptor *fd* into a mutable buffer object *buffer*.

The *buffer* should be mutable and *bytes-like*. On success, returns the number of bytes read. Less bytes may be read than the size of the buffer. The underlying system call will be retried when interrupted by a signal, unless the signal handler raises an exception. Other errors will not be retried and an error will be raised.

Returns 0 if *fd* is at end of file or if the provided *buffer* has length 0 (which can be used to check for errors without reading data). Never returns negative.

> **Note**
>
> This function is intended for low-level I/O and must be applied to a file descriptor as returned by `os.open()` or `os.pipe()`. To read a "file object" returned by the built-in function `open()`, or `sys.stdin`, use its member functions, for example `io.BufferedIOBase.readinto()`, `io.BufferedIOBase.read()`, or `io.TextIOBase.read()`

Ajouté dans la version 3.14.

`os.sendfile`(*out_fd*, *in_fd*, *offset*, *count*)

`os.sendfile`(*out_fd*, *in_fd*, *offset*, *count*, *headers=()*, *trailers=()*, *flags=0*)

Copy *count* bytes from file descriptor *in_fd* to file descriptor *out_fd* starting at *offset*. Return the number of bytes sent. When EOF is reached return `0`.

La première notation de fonction est prise en charge par toutes les plate-formes qui définissent `sendfile()`.

On Linux, if *offset* is given as `None`, the bytes are read from the current position of *in_fd* and the position of *in_fd* is updated.

The second case may be used on macOS and FreeBSD where *headers* and *trailers* are arbitrary sequences of buffers that are written before and after the data from *in_fd* is written. It returns the same as the first case.

On macOS and FreeBSD, a value of `0` for *count* specifies to send until the end of *in_fd* is reached.

All platforms support sockets as *out_fd* file descriptor, and some platforms allow other types (e.g. regular file, pipe) as well.

Les applications multiplateformes ne devraient pas utiliser les arguments *headers*, *trailers*, et *flags*.

Availability : Unix, not WASI.

> **Note**
>
> Pour une interface de plus haut niveau de `sendfile()`, voir `socket.socket.setfile()`.

Ajouté dans la version 3.3.

Modifié dans la version 3.9 : Les paramètres *out* et *in* ont été renommés *out_fd* et *in_fd*.

os.**SF_NODISKIO**

os.**SF_MNOWAIT**

os.**SF_SYNC**

Paramètres de la fonction `sendfile()`, si l'implémentation les gère.

Availability : Unix, not WASI.

Ajouté dans la version 3.3.

os.**SF_NOCACHE**

Parameter to the `sendfile()` function, if the implementation supports it. The data won't be cached in the virtual memory and will be freed afterwards.

Availability : Unix, not WASI.

Ajouté dans la version 3.11.

os.**set_blocking**(*fd*, *blocking*, /)

Définit le mode bloquant d'un descripteur de fichier spécifié. Assigne l'indicateur `O_NONBLOCK` si *blocking* vaut `False`, efface l'indicateur sinon.

Voir aussi `get_blocking()` et `socket;socket.setblocking()`.

Availability : Unix, Windows.

The function is limited on WASI, see *Plateformes WebAssembly* for more information.

On Windows, this function is limited to pipes.

Ajouté dans la version 3.5.

Modifié dans la version 3.12 : Added support for pipes on Windows.

os.**splice**(*src*, *dst*, *count*, *offset_src=None*, *offset_dst=None*, *flags=0*)

Transfer *count* bytes from file descriptor *src*, starting from offset *offset_src*, to file descriptor *dst*, starting from offset *offset_dst*.

The splicing behaviour can be modified by specifying a *flags* value. Any of the following variables may used, combined using bitwise OR (the `|` operator) :

— If `SPLICE_F_MOVE` is specified, the kernel is asked to move pages instead of copying, but pages may still be copied if the kernel cannot move the pages from the pipe.

— If `SPLICE_F_NONBLOCK` is specified, the kernel is asked to not block on I/O. This makes the splice pipe operations nonblocking, but splice may nevertheless block because the spliced file descriptors may block.

— If `SPLICE_F_MORE` is specified, it hints to the kernel that more data will be coming in a subsequent splice.

At least one of the file descriptors must refer to a pipe. If *offset_src* is `None`, then *src* is read from the current position ; respectively for *offset_dst*. The offset associated to the file descriptor that refers to a pipe must be `None`. The files pointed to by *src* and *dst* must reside in the same filesystem, otherwise an `OSError` is raised with `errno` set to `errno.EXDEV`.

Cette copie est faite sans le coût additionnel de transférer les données depuis le noyau vers l'espace utilisateur puis dans le sens inverse vers le noyau. En outre, certains systèmes de fichiers peuvent implémenter des optimisations supplémentaires. Cette copie est faite comme si les deux fichiers étaient ouverts en mode binaire.

Upon successful completion, returns the number of bytes spliced to or from the pipe. A return value of 0 means end of input. If *src* refers to a pipe, then this means that there was no data to transfer, and it would not make sense to block because there are no writers connected to the write end of the pipe.

Voir aussi

The `splice(2)` man page.

Availability : Linux >= 2.6.17 with glibc >= 2.5

Ajouté dans la version 3.10.

os.**SPLICE_F_MOVE**

os.**SPLICE_F_NONBLOCK**

os.**SPLICE_F_MORE**

Ajouté dans la version 3.10.

os.**readv**(*fd*, *buffers*, /)

Lit depuis un descripteur de fichier *fd* dans une séquence d'*objets bytes-compatibles* mutables : *buffers*. Les données sont transférées dans chaque tampon, jusqu'à ce qu'il soit plein, tour à tour.

Renvoie le nombre total d'octets réellement lus, qui peut être inférieur à la capacité totale de tous les objets.

Le système d'exploitation peut définir une limite (valeur `sysconf()` `'SC_IOV_MAX'`) sur le nombre de mémoires tampons pouvant être utilisées.

Availability : Unix.

Ajouté dans la version 3.3.

os.**tcgetpgrp**(*fd*, /)

Renvoie le groupe de processus associé au terminal donné par *fd* (un descripteur de fichier ouvert comme renvoyé par `os.open()`).

Availability : Unix, not WASI.

os.**tcsetpgrp**(*fd*, *pg*, /)

Place *pg* dans le groupe de processus associé au terminal donné par *fd* (un descripteur de fichier ouvert comme renvoyé par `os.open()`).

Availability : Unix, not WASI.

os.**ttyname**(*fd*, /)

Renvoie une chaîne de caractères spécifiant le périphérique terminal associé au descripteur de fichier *fd*. Si *fd* n'est pas associé à un périphérique terminal, une exception est levée.

Availability : Unix.

os.**unlockpt**(*fd*, /)

Unlock the slave pseudo-terminal device associated with the master pseudo-terminal device to which the file descriptor *fd* refers. The file descriptor *fd* is not closed upon failure.

Calls the C standard library function `unlockpt()`.

Availability : Unix, not WASI.

Ajouté dans la version 3.13.

os.**write**(*fd*, *str*, /)

Écrit la chaîne d'octets de *str* vers le descripteur de fichier *fd*.

Renvoie le nombre d'octets effectivement écrits.

> **Note**
>
> Cette fonction est destinée aux entrées-sorties bas niveau et doit être appliquée à un descripteur de fichier comme renvoyé par `os.open()` ou `pipe()`. Pour écrire dans un « fichier objet » renvoyé par la primitive `open()`, `popen()`, ou par `fdopen()`, ou par `sys.stdout` ou `sys.stderr`, utilisez sa méthode `write()`.

Modifié dans la version 3.5 : Si l'appel système est interrompu et que le gestionnaire de signal ne lève aucune exception, la fonction réessaye l'appel système au lieu de lever une `InterruptedError` (voir la **PEP 475** à propos du raisonnement).

os.**writev**(*fd*, *buffers*, /)

Écrit le contenu de *buffers* vers le descripteur de fichier *fd*. *buffers* doit être une séquence d'*objets bytes-compatibles*. Les tampons sont traités dans l'ordre du tableau. Le contenu entier du premier tampon est écrit avant le traitement du second, etc.

Renvoie le nombre total d'octets effectivement écrits.

Le système d'exploitation peut définir une limite (valeur `sysconf()` `'SC_IOV_MAX'`) sur le nombre de mémoires tampons pouvant être utilisées.

Availability : Unix.

Ajouté dans la version 3.3.

Demander la taille d'un terminal

Ajouté dans la version 3.3.

os.**get_terminal_size**(*fd=STDOUT_FILENO*, /)

Renvoie la taille du terminal comme un couple `(columns, lines)` de type `terminal_size`.

L'argument optionnel `fd` (par défaut : `STDOUT_FILENO`, ou la sortie standard) spécifie le descripteur de fichier auquel la requête doit être envoyée.

Si le descripteur de fichier n'est pas connecté à un terminal, une `OSError` est levée.

`shutil.get_terminal_size()` est la fonction haut-niveau qui devrait normalement être utilisée, `os.get_terminal_size` en est l'implémentation bas-niveau.

Availability : Unix, Windows.

class os.**terminal_size**

Une sous-classe de *tuple*, contenant `(columns, lines)`, la taille du terminal.

columns

Longueur du terminal en caractères.

lines

Hauteur du terminal en caractères.

Héritage de descripteurs de fichiers

Ajouté dans la version 3.4.

Un descripteur de fichier a un indicateur indiquant s'il peut être hérité par les processus-fils. Depuis Python 3.4, les descripteurs de fichiers créés par Python ne sont pas héritable par défaut.

Sur UNIX, les descripteurs de fichiers non-héritables sont fermés dans les processus-fils à l'exécution, les autres descripteurs sont hérités.

Sur Windows, les fichiers et identificateurs non-héritables sont fermés dans les processus-fils, à part les flux standards (descripteurs 0, 1, et 2 : *stdin*, *stdout* et *stderr*) qui sont toujours hérités. En utilisant les fonctions `spawn*`, tous les identificateurs héritables et les descripteurs de fichiers héritables sont hérités. En utilisant le module `subprocess`, tous les descripteurs de fichiers (à part les flux standards) sont fermés, et les identificateurs héritables sont hérités seulement si le paramètre *close_fds* vaut `False`.

On WebAssembly platforms, the file descriptor cannot be modified.

os.**get_inheritable**(*fd*, /)

Récupère le marqueur « héritable » (booléen) du descripteur de fichier spécifié.

os.**set_inheritable**(*fd*, *inheritable*, /)

Définit le marqueur « héritable » du descripteur de fichier spécifié.

os.**get_handle_inheritable**(*handle*, /)

Récupère le marqueur « héritable » (booléen) de l'identificateur spécifié.

Availability : Windows.

os.**set_handle_inheritable**(*handle*, *inheritable*, /)

Définit le marqueur « héritable » de l'identificateur spécifié.

Availability : Windows.

16.1.6 Fichiers et répertoires

Sur certaines plate-formes Unix, beaucoup de ces fonctions gèrent une ou plusieurs des fonctionnalités suivantes :

— **specifying a file descriptor :** Normally the *path* argument provided to functions in the `os` module must be a string specifying a file path. However, some functions now alternatively accept an open file descriptor for their *path* argument. The function will then operate on the file referred to by the descriptor. (For POSIX systems, Python will call the variant of the function prefixed with `f` (e.g. call `fchdir` instead of `chdir`).)

You can check whether or not *path* can be specified as a file descriptor for a particular function on your platform using `os.supports_fd`. If this functionality is unavailable, using it will raise a `NotImplementedError`.

If the function also supports *dir_fd* or *follow_symlinks* arguments, it's an error to specify one of those when supplying *path* as a file descriptor.

— **paths relative to directory descriptors :** If *dir_fd* is not `None`, it should be a file descriptor referring to a directory, and the path to operate on should be relative ; path will then be relative to that directory. If the path is absolute, *dir_fd* is ignored. (For POSIX systems, Python will call the variant of the function with an `at` suffix and possibly prefixed with `f` (e.g. call `faccessat` instead of `access`).

You can check whether or not *dir_fd* is supported for a particular function on your platform using `os.supports_dir_fd`. If it's unavailable, using it will raise a `NotImplementedError`.

— **not following symlinks :** If *follow_symlinks* is `False`, and the last element of the path to operate on is a symbolic link, the function will operate on the symbolic link itself rather than the file pointed to by the link. (For POSIX systems, Python will call the `l...` variant of the function.)

You can check whether or not *follow_symlinks* is supported for a particular function on your platform using `os.supports_follow_symlinks`. If it's unavailable, using it will raise a `NotImplementedError`.

os.**access**(*path*, *mode*, *, *dir_fd=None*, *effective_ids=False*, *follow_symlinks=True*)

Utilise l'*uid*/*gid* réel pour tester l'accès à *path*. Notez que la plupart des opérations utiliseront l'*uid*/*gid* effectif, dès lors cette méthode peut être utilisée dans un environnement *suid*/*sgid* pour tester si l'utilisateur invoquant a les droits d'accès pour accéder à *path*. *mode* devrait être `F_OK` pour tester l'existence de *path*, ou il peut être le OR (OU inclusif) d'une ou plusieurs des constantes suivantes : `R_OK`, `W_OK`, et `X_OK` pour tester les permissions. Renvoie `True` si l'accès est permis, et `False` s'il ne l'est pas. Voir la page de manuel Unix `access(2)` pour plus d'informations.

Cette fonction peut gérer la spécification de *chemins relatifs vers des descripteurs de fichiers* et *le suivi des liens symboliques*.

Si *effective_id* vaut `True`, `access()` effectuera ses vérifications d'accès en utilisant l'*uid*/*gid* effectif à la place de l'*uid*/*gid* réel. *effective_ids* peut ne pas être géré sur votre plate-forme, vous pouvez vérifier s'il est disponible en utilisant `os.supports_effective_ids`. S'il est indisponible, l'utiliser lèvera une `NotImplementedError`.

> **Note**
>
> Utiliser `access()` pour vérifier si un utilisateur est autorisé (par exemple) à ouvrir un fichier avant d'effectivement le faire en utilisant `open()` crée une faille de sécurité : l'utilisateur peut exploiter le court intervalle de temps entre la vérification et l'ouverture du fichier pour le manipuler. Il est préférable d'utiliser les techniques *EAFP*. Par exemple :

```
if os.access("myfile", os.R_OK):
    with open("myfile") as fp:
        return fp.read()
return "some default data"
```

est mieux écrit comme suit :

```
try:
    fp = open("myfile")
except PermissionError:
    return "some default data"
else:
    with fp:
        return fp.read()
```

Note

Les opérations d'entrées/sorties peuvent échouer même quand `access()` indique qu'elles devraient réussir, particulièrement pour les opérations sur les systèmes de fichiers réseaux qui peuvent avoir une sémantique de permissions au-delà du modèle de bits de permission usuel POSIX.

Modifié dans la version 3.3 : Paramètres *dir_fd*, *effective_ids*, et *follow_symlinks* ajoutés.

Modifié dans la version 3.6 : Accepte un *path-like object*.

`os.F_OK`

`os.R_OK`

`os.W_OK`

`os.X_OK`

Valeurs à passer au paramètre *mode* de `access()` pour tester respectivement l'existence, les droits de lecture, d'écriture et d'exécution.

`os.chdir(path)`

Change le répertoire de travail actuel par *path*.

Cette fonction prend en charge *la spécification d'un descripteur de fichier*. Le descripteur doit référencer un répertoire ouvert, pas un fichier ouvert.

Cette fonction peut lever `OSError` et des sous-classes telles que `FileNotFoundError`, `PermissionError` et `NotADirectoryError`.

Lève un *événement d'audit* `os.chdir` avec l'argument `path`.

Modifié dans la version 3.3 : Prise en charge de la spécification de *path* par un descripteur de fichier sur certaines plate-formes.

Modifié dans la version 3.6 : Accepte un *path-like object*.

`os.chflags(path, flags, *, follow_symlinks=True)`

Définit les marqueurs de *path* par la valeur numérique *flags*. *flags* peut prendre une combinaison (OU bit-à-bit) des valeurs suivantes (comme défini dans le module `stat`) :

- `stat.UF_NODUMP`
- `stat.UF_IMMUTABLE`
- `stat.UF_APPEND`
- `stat.UF_OPAQUE`
- `stat.UF_NOUNLINK`
- `stat.UF_COMPRESSED`
- `stat.UF_HIDDEN`
- `stat.SF_ARCHIVED`
- `stat.SF_IMMUTABLE`

— *stat.SF_APPEND*
— *stat.SF_NOUNLINK*
— *stat.SF_SNAPSHOT*

Cette fonction prend en charge *le suivi des liens symboliques*.

Lève un *événement d'audit* `os.chflags` avec les arguments `path`, `flags`.

Availability : Unix, not WASI.

Modifié dans la version 3.3 : Added the *follow_symlinks* parameter.

Modifié dans la version 3.6 : Accepte un *path-like object*.

`os.chmod`(*path*, *mode*, *, *dir_fd=None*, *follow_symlinks=True*)

Change le mode de *path* par la valeur numérique *mode*. *mode* peut prendre une des valeurs suivantes (comme défini dans le module `stat`) ou une combinaison (OU bit-à-bit) de ces valeurs :

— *stat.S_ISUID*
— *stat.S_ISGID*
— *stat.S_ENFMT*
— *stat.S_ISVTX*
— *stat.S_IREAD*
— *stat.S_IWRITE*
— *stat.S_IEXEC*
— *stat.S_IRWXU*
— *stat.S_IRUSR*
— *stat.S_IWUSR*
— *stat.S_IXUSR*
— *stat.S_IRWXG*
— *stat.S_IRGRP*
— *stat.S_IWGRP*
— *stat.S_IXGRP*
— *stat.S_IRWXO*
— *stat.S_IROTH*
— *stat.S_IWOTH*
— *stat.S_IXOTH*

Cette fonction prend en charge *la spécification d'un descripteur de fichier*, *les chemins relatifs à des descripteurs de répertoires*, et *le non-suivi des liens symboliques*.

> **Note**
>
> Although Windows supports `chmod()`, you can only set the file's read-only flag with it (via the `stat.S_IWRITE` and `stat.S_IREAD` constants or a corresponding integer value). All other bits are ignored. The default value of *follow_symlinks* is `False` on Windows.
>
> The function is limited on WASI, see *Plateformes WebAssembly* for more information.

Lève un *événement d'audit* `os.chmod` avec les arguments `path`, `mode`, `dir_fd`.

Modifié dans la version 3.3 : Prise en charge de la spécification de *path* par un répertoire ouvert et des arguments *dir_fd* et *follow_symlinks* ajoutés.

Modifié dans la version 3.6 : Accepte un *path-like object*.

Modifié dans la version 3.13 : Added support for a file descriptor and the *follow_symlinks* argument on Windows.

`os.chown`(*path*, *uid*, *gid*, *, *dir_fd=None*, *follow_symlinks=True*)

Change l'identifiant du propriétaire et du groupe de *path* par les valeurs numériques *uid* et *gid*. Pour laisser l'un de ces identifiants inchangé, le définir à `-1`.

Cette fonction prend en charge *la spécification d'un descripteur de fichier*, *les chemins relatifs à des descripteurs de répertoires*, et *le non-suivi des liens symboliques*.

Voir `shutil.chown()` pour une fonction de plus haut-niveau qui accepte des noms en plus des identifiants numériques.

Lève un *événement d'audit* `os.chown` avec les arguments `path`, `uid`, `gid`, `dir_fd`.

Availability : Unix.

The function is limited on WASI, see *Plateformes WebAssembly* for more information.

Modifié dans la version 3.3 : Prise en charge de la spécification de *path* par un répertoire ouvert et des arguments *dir_fd* et *follow_symlinks* ajoutés.

Modifié dans la version 3.6 : Accepte un *path-like object*.

`os.chroot(path)`

Change le répertoire racine du processus actuel par *path*.

Availability : Unix, not WASI, not Android.

Modifié dans la version 3.6 : Accepte un *path-like object*.

`os.fchdir(fd)`

Change le répertoire de travail actuel par le répertoire représenté par le descripteur de fichier *fd*. Le descripteur doit référencer un répertoire ouvert, pas un fichier ouvert. Depuis Python 3.3, c'est équivalent à `os.chdir(fd)`.

Lève un *événement d'audit* `os.chdir` avec l'argument `path`.

Availability : Unix.

`os.getcwd()`

Renvoie une chaîne de caractères représentant le répertoire de travail actuel.

`os.getcwdb()`

Renvoie une chaîne de *bytes* représentant le répertoire de travail actuel.

Modifié dans la version 3.8 : La fonction utilise maintenant l'encodage UTF-8 sur Windows, plutôt que la *page de code* ANSI : la **PEP 529** explique la raison. Cette fonction n'est plus obsolète sur Windows.

`os.lchflags(path, flags)`

Définit les marqueurs de *path* par la valeur numérique *flags*, comme `chflags()`, mais ne suit pas les liens symboliques. Depuis Python 3.3, c'est équivalent à `os.chflags(path, flags, follow_symlinks=False)`.

Lève un *événement d'audit* `os.chflags` avec les arguments `path`, `flags`.

Availability : Unix, not WASI.

Modifié dans la version 3.6 : Accepte un *path-like object*.

`os.lchmod(path, mode)`

Change le mode de *path* par la valeur numérique *mode*. Si *path* est un lien symbolique, ça affecte le lien symbolique à la place de la cible. Voir la documentation pour les valeurs possibles de *mode* pour `chmod()`. Depuis Python 3.3, c'est équivalent à `os.chmod(path, mode, follow_symlinks=False)`.

`lchmod()` is not part of POSIX, but Unix implementations may have it if changing the mode of symbolic links is supported.

Lève un *événement d'audit* `os.chmod` avec les arguments `path`, `mode`, `dir_fd`.

Availability : Unix, Windows, not Linux, FreeBSD >= 1.3, NetBSD >= 1.3, not OpenBSD

Modifié dans la version 3.6 : Accepte un *path-like object*.

Modifié dans la version 3.13 : Added support on Windows.

`os.lchown(path, uid, gid)`

Change les identifiants du propriétaire et du groupe de *path* par *uid* et *gid*. Cette fonction ne sui-

vra pas les liens symboliques. Depuis Python 3.3, c'est équivalent à `os.chown(path, uid, gid, follow_symlinks=False)`.

Lève un *événement d'audit* `os.chown` avec les arguments `path`, `uid`, `gid`, `dir_fd`.

Availability : Unix.

Modifié dans la version 3.6 : Accepte un *path-like object*.

`os.link`(*src*, *dst*, *, *src_dir_fd=None*, *dst_dir_fd=None*, *follow_symlinks=True*)

Crée un lien matériel appelé *dst* pointant sur *src*.

Cette fonction prend en charge la spécification *src_dir_fd* et/ou *dst_dir_fd* pour préciser *des chemins relatifs à des descripteurs de répertoires*, et *le non-suivi des liens symboliques*.

Lève un *événement d'audit* `os.link` avec les arguments `src`, `dst`, `src_dir_ds`, `dst_dir_fd`.

Availability : Unix, Windows.

Modifié dans la version 3.2 : Prise en charge de Windows.

Modifié dans la version 3.3 : Added the *src_dir_fd*, *dst_dir_fd*, and *follow_symlinks* parameters.

Modifié dans la version 3.6 : Accepte un *path-like object* pour *src* et *dst*.

`os.listdir`(*path='.'*)

Return a list containing the names of the entries in the directory given by *path*. The list is in arbitrary order, and does not include the special entries `'.'` and `'..'` even if they are present in the directory. If a file is removed from or added to the directory during the call of this function, whether a name for that file be included is unspecified.

path peut être un *path-like object*. Si *path* est de type `bytes` (directement ou indirectement à travers une interface `PathLike`), les noms de fichiers renvoyés seront aussi de type `bytes` ; dans toutes les autres circonstances, ils seront de type `str`.

Cette fonction peut également gérer *la spécification de descripteurs de fichiers*. Le descripteur doit référencer un répertoire.

Lève un *événement d'audit* `os.listdir` avec l'argument `path`.

Note

Pour encoder des noms de fichiers de type `str` en `bytes`, utilisez la fonction `encode()`.

Voir aussi

La fonction `scandir()` renvoie les entrées du répertoire ainsi que leurs attributs , offrant une meilleure performance pour beaucoup de cas utilisés fréquemment.

Modifié dans la version 3.2 : Le paramètre *path* est devenu optionnel.

Modifié dans la version 3.3 : Ajout de la possibilité de spécifier *path* comme descripteur de fichier ouvert.

Modifié dans la version 3.6 : Accepte un *path-like object*.

`os.listdrives`()

Return a list containing the names of drives on a Windows system.

A drive name typically looks like `'C:\\'`. Not every drive name will be associated with a volume, and some may be inaccessible for a variety of reasons, including permissions, network connectivity or missing media. This function does not test for access.

May raise `OSError` if an error occurs collecting the drive names.

Raises an *auditing event* `os.listdrives` with no arguments.

Availability : Windows

Ajouté dans la version 3.12.

`os.listmounts(volume)`

Return a list containing the mount points for a volume on a Windows system.

volume must be represented as a GUID path, like those returned by `os.listvolumes()`. Volumes may be mounted in multiple locations or not at all. In the latter case, the list will be empty. Mount points that are not associated with a volume will not be returned by this function.

The mount points return by this function will be absolute paths, and may be longer than the drive name.

Raises `OSError` if the volume is not recognized or if an error occurs collecting the paths.

Raises an *auditing event* `os.listmounts` with argument `volume`.

Availability : Windows

Ajouté dans la version 3.12.

`os.listvolumes()`

Return a list containing the volumes in the system.

Volumes are typically represented as a GUID path that looks like `\\?\Volume{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}\`. Files can usually be accessed through a GUID path, permissions allowing. However, users are generally not familiar with them, and so the recommended use of this function is to retrieve mount points using `os.listmounts()`.

May raise `OSError` if an error occurs collecting the volumes.

Raises an *auditing event* `os.listvolumes` with no arguments.

Availability : Windows

Ajouté dans la version 3.12.

`os.lstat(path, *, dir_fd=None)`

Perform the equivalent of an `lstat()` system call on the given path. Similar to `stat()`, but does not follow symbolic links. Return a `stat_result` object.

Sur les plate-formes qui ne gérant pas les liens symboliques, c'est un alias pour `stat()`.

Depuis Python 3.3, c'est équivalent à `os.stat(path, dir_fd=dir_fd, follow_symlinks=False)`.

Cette fonction peut également gérer *des chemins relatifs à des descripteurs de répertoires*.

Voir aussi

La fonction `stat()`.

Modifié dans la version 3.2 : Prise en charge des les liens symboliques sur Windows 6.0 (Vista).

Modifié dans la version 3.3 : Paramètre *dir_fd* ajouté.

Modifié dans la version 3.6 : Accepte un *path-like object*.

Modifié dans la version 3.8 : On Windows, now opens reparse points that represent another path (name surrogates), including symbolic links and directory junctions. Other kinds of reparse points are resolved by the operating system as for `stat()`.

`os.mkdir(path, mode=0o777, *, dir_fd=None)`

Crée un répertoire appelé *path* avec pour mode, la valeur numérique *mode*.

If the directory already exists, `FileExistsError` is raised. If a parent directory in the path does not exist, `FileNotFoundError` is raised.

Sous certains systèmes, *mode* est ignoré. Quand il est utilisé, il lui est premièrement appliqué le masque courant

umask. Si des bits autres que les 9 derniers sont activés (c.-à-d. les 3 derniers chiffres de la représentation octale de *mode*), leur signification sera dépendante de la plate-forme. Sous certaines plate-formes, ils seront ignorés et vous devrez appeler explicitement `chmod()` pour les modifier.

On Windows, a *mode* of `0o700` is specifically handled to apply access control to the new directory such that only the current user and administrators have access. Other values of *mode* are ignored.

Cette fonction peut également gérer *des chemins relatifs à des descripteurs de répertoires*.

Il est également possible de créer des répertoires temporaires, voir la fonction `tempfile.mkdtemp()` du module `tempfile`.

Lève un *événement d'audit* `os.mkdir` avec les arguments `path`, `mode`, `dir_fd`.

Modifié dans la version 3.3 : Paramètre *dir_fd* ajouté.

Modifié dans la version 3.6 : Accepte un *path-like object*.

Modifié dans la version 3.13 : Windows now handles a *mode* of `0o700`.

`os.`**`makedirs`**(*name*, *mode=0o777*, *exist_ok=False*)

Fonction de création récursive de répertoires. Comme `mkdir()` mais crée tous les répertoires de niveau intermédiaire nécessaires pour contenir le répertoire « feuille ».

The *mode* parameter is passed to `mkdir()` for creating the leaf directory ; see *the mkdir() description* for how it is interpreted. To set the file permission bits of any newly created parent directories you can set the umask before invoking `makedirs()`. The file permission bits of existing parent directories are not changed.

If *exist_ok* is `False` (the default), a `FileExistsError` is raised if the target directory already exists.

> **Note**
>
> Un appel à `makedirs()` est confus si les éléments du chemin à créer contiennent `pardir` (par exemple, `"..."` sur les systèmes UNIX).

Cette fonction gère les chemins UNC correctement.

Lève un *événement d'audit* `os.mkdir` avec les arguments `path`, `mode`, `dir_fd`.

Modifié dans la version 3.2 : Added the *exist_ok* parameter.

Modifié dans la version 3.4.1 : Avant Python 3.4.1, si *exist_ok* valait `True` et le répertoire à créer existait, `makedirs()` aurait levé une erreur si *mode* n'était pas équivalent au mode du répertoire existant. Puisque ce comportement n'était pas possible) implémenter de manière sécurisée, il a été retiré pour Python 3.4.1. Voir bpo-21082.

Modifié dans la version 3.6 : Accepte un *path-like object*.

Modifié dans la version 3.7 : The *mode* argument no longer affects the file permission bits of newly created intermediate-level directories.

`os.`**`mkfifo`**(*path*, *mode=0o666*, *, *dir_fd=None*)

Crée un FIFO (*First In, First Out*, ou un tube (*pipe* en anglais) nommé) appelé *path* avec le mode numérique *mode*. La valeur actuelle de *umask* est d'abord masquée du mode.

Cette fonction peut également gérer *des chemins relatifs à des descripteurs de répertoires*.

Les FIFOs sont des tubes qui peuvent être accédés comme des fichiers normaux. Les FIFOs existent jusqu'à ce qu'ils soient retirés (par exemple, à l'aide de `os.unlink()`). Généralement, les FIFOs sont utilisé comme communication entre des processus de type « client » et « serveur » : le serveur ouvre le FIFO pour le lire, et le client l'ouvre pour écrire dedans. Notez que `mkfifo()` n'ouvre pas le FIFO — il crée juste un point de rendez-vous.

Availability : Unix, not WASI.

Modifié dans la version 3.3 : Paramètre *dir_fd* ajouté.

Modifié dans la version 3.6 : Accepte un *path-like object*.

os.**mknod**(*path*, *mode=0o600*, *device=0*, *, *dir_fd=None*)

Crée un nœud du système de fichiers (fichier, périphérique, fichier spécial, ou tuyau nommé) appelée *path*. *mode* spécifie à la fois les permissions à utiliser et le type de nœud à créer, en étant combiné (OR bit-à-bit) avec l'une des valeurs suivantes : `stat.S_IFREG`, `stat.S_IFCHR`, `stat.S_IFBLK`, et `stat.S_IFIFO` (ces constantes sont disponibles dans le module `stat`). Pour `stat.S_IFCHR` et `stat.S_IFBLK`, *device* définit le fichier spécial de périphérique tout juste créé (probablement en utilisant `os.makedev()`), sinon, cet argument est ignoré.

Cette fonction peut également gérer *des chemins relatifs à des descripteurs de répertoires*.

Availability : Unix, not WASI.

Modifié dans la version 3.3 : Paramètre *dir_fd* ajouté.

Modifié dans la version 3.6 : Accepte un *path-like object*.

os.**major**(*device*, /)

Extract the device major number from a raw device number (usually the `st_dev` or `st_rdev` field from `stat`).

os.**minor**(*device*, /)

Extract the device minor number from a raw device number (usually the `st_dev` or `st_rdev` field from `stat`).

os.**makedev**(*major*, *minor*, /)

Compose un nombre de périphérique brut à partir des nombres de périphérique mineur et majeur.

os.**pathconf**(*path*, *name*)

Renvoie des informations sur la configurations relatives à un fichier déterminé. *name* spécifie la valeur de configuration à récupérer ; ce peut être une chaîne de caractères qui est le nom d'une valeur système particulière. Ces noms sont spécifiés dans certains standards (POSIX.1, Unix 95, Unix 98, etc). Certaines plate-formes définissent des noms supplémentaires également. Les noms connus du système d'exploitation hôte sont donnés dans le dictionnaire `pathconf_names`. Pour les variables de configuration non incluses dans ce *mapping*, passer un entier pour *name* est également accepté.

Si *name* est une chaîne de caractères et n'est pas connu, une `ValueError` est levée. Si une valeur spécifique de *name* n'est pas gérée par le système hôte, même si elle est incluse dans `pathconf_names`, une `OSError` est levée avec `errno.EINVAL` pour code d'erreur.

Cette fonction prend en charge *la spécification d'un descripteur de fichier*.

Availability : Unix.

Modifié dans la version 3.6 : Accepte un *path-like object*.

os.**pathconf_names**

Dictionnaire liant les noms acceptés par les fonctions `pathconf()` et `fpathconf()` aux valeurs entières définies pour ces noms par le système d'exploitation hôte. Cette variable peut être utilisée pour déterminer l'ensemble des noms connus du système d'exploitation.

Availability : Unix.

os.**readlink**(*path*, *, *dir_fd=None*)

Renvoie une chaîne de caractères représentant le chemin vers lequel le lien symbolique pointe. Le résultat peut être soit un chemin relatif, soit un chemin absolu. S'il est relatif, il peut être converti en chemin absolu en utilisant `os.path.join(os.path.dirname(path), result)`.

Si *path* est une chaîne de caractères (directement ou indirectement à travers une interface `PathLike`), le résultat sera aussi une chaîne de caractères, et l'appel pourra lever une UnicodeDecodeError. Si *path* est une chaîne d'octets (directement ou indirectement), le résultat sera une chaîne d'octets.

Cette fonction peut également gérer *des chemins relatifs à des descripteurs de répertoires*.

Lorsque vous essayez de résoudre un chemin qui peut contenir des liens, utilisez `realpath()` pour gérer correctement la récursion et les différences de plate-forme.

Availability : Unix, Windows.

Modifié dans la version 3.2 : Prise en charge des les liens symboliques sur Windows 6.0 (Vista).

Modifié dans la version 3.3 : Paramètre *dir_fd* ajouté.

Modifié dans la version 3.6 : Accepte un *path-like object* sur Unix.

Modifié dans la version 3.8 : Accepte un *path-like object* et une chaine d'octets sous Windows.

Added support for directory junctions, and changed to return the substitution path (which typically includes `\\?\` prefix) rather than the optional "print name" field that was previously returned.

`os.remove`(*path*, *, *dir_fd=None*)

Remove (delete) the file *path*. If *path* is a directory, an `OSError` is raised. Use `rmdir()` to remove directories. If the file does not exist, a `FileNotFoundError` is raised.

Cette fonction prend en charge *des chemins relatifs à des descripteurs de répertories*.

Sur Windows, tenter de retirer un fichier en cours d'utilisation cause la levée d'une exception, sur Unix, l'entrée du répertoire est supprimé mais l'espace de stockage alloué au fichier ne sera pas disponible avant que le fichier original ne soit plus utilisé.

La fonction est sémantiquement identique à `unlink()`.

Lève un *événement d'audit* `os.remove` avec les arguments `path`, `dir_fd`.

Modifié dans la version 3.3 : Paramètre *dir_fd* ajouté.

Modifié dans la version 3.6 : Accepte un *path-like object*.

`os.removedirs`(*name*)

Supprime des répertoires récursivement. Fonctionne comme `rmdir()` si ce n'est que si le répertoire feuille est retiré avec succès, `removedirs()` essaye de supprimer successivement chaque répertoire parent mentionné dans *path* jusqu'à ce qu'un erreur ne soit levée (ce qui est ignoré car la signification générale en est qu'un répertoire parent n'est pas vide). Par exemple, `os.removedirs('foo/bar/baz')` supprimera d'abord le répertoire `'foo/bar/baz'`, et ensuite supprimera `'foo/bar'` et puis `'foo'` s'ils sont vides. Lève une `OSError` si le répertoire feuille n'a pas pu être supprimé avec succès.

Lève un *événement d'audit* `os.remove` avec les arguments `path`, `dir_fd`.

Modifié dans la version 3.6 : Accepte un *path-like object*.

`os.rename`(*src*, *dst*, *, *src_dir_fd=None*, *dst_dir_fd=None*)

Renomme le fichier ou le répertoire *src* en *dst*. Si *dst* existe, l'opération échoue avec une sous-classe `OSError` dans un certain nombre de cas :

On Windows, if *dst* exists a `FileExistsError` is always raised. The operation may fail if *src* and *dst* are on different filesystems. Use `shutil.move()` to support moves to a different filesystem.

On Unix, if *src* is a file and *dst* is a directory or vice-versa, an `IsADirectoryError` or a `NotADirectoryError` will be raised respectively. If both are directories and *dst* is empty, *dst* will be silently replaced. If *dst* is a non-empty directory, an `OSError` is raised. If both are files, *dst* will be replaced silently if the user has permission. The operation may fail on some Unix flavors if *src* and *dst* are on different filesystems. If successful, the renaming will be an atomic operation (this is a POSIX requirement).

Cette fonction prend en charge les spécifications *src_dir_fd* et/ou *dst_dir_fd* pour fournir *des chemins relatifs à des descripteurs de fichiers*.

Si vous désirez un écrasement multiplateformes de la destination, utilisez la fonction `replace()`.

Lève un *événement d'audit* `os.rename` avec les arguments `src`, `dst`, `src_dir_ds`, `dst_dir_fd`.

Modifié dans la version 3.3 : Added the *src_dir_fd* and *dst_dir_fd* parameters.

Modifié dans la version 3.6 : Accepte un *path-like object* pour *src* et *dst*.

`os.renames`(*old*, *new*)

Fonction récursive de renommage de fichiers ou répertoires. Fonctionne comme `rename()`, si ce n'est que la

création d'un répertoire intermédiaire nécessaire pour rendre le nouveau chemin correct est essayé en premier. Après le renommage, les répertoires correspondant aux segments de chemin les plus à droite de l'ancien nom seront élagués en utilisant `removedirs()`.

> **Note**
>
> Cette fonction peut échouer avec la nouvelle structure de dictionnaire définie si vous n'avez pas les permissions nécessaires pour supprimer le répertoire ou fichier feuille.

Lève un *événement d'audit* `os.rename` avec les arguments `src`, `dst`, `src_dir_ds`, `dst_dir_fd`.

Modifié dans la version 3.6 : Accepte un *path-like object* pour *old* et *new*.

`os.replace`(*src*, *dst*, *, *src_dir_fd=None*, *dst_dir_fd=None*)

Rename the file or directory *src* to *dst*. If *dst* is a non-empty directory, `OSError` will be raised. If *dst* exists and is a file, it will be replaced silently if the user has permission. The operation may fail if *src* and *dst* are on different filesystems. If successful, the renaming will be an atomic operation (this is a POSIX requirement).

Cette fonction prend en charge les spécifications *src_dir_fd* et/ou *dst_dir_fd* pour fournir *des chemins relatifs à des descripteurs de fichiers*.

Lève un *événement d'audit* `os.rename` avec les arguments `src`, `dst`, `src_dir_ds`, `dst_dir_fd`.

Ajouté dans la version 3.3.

Modifié dans la version 3.6 : Accepte un *path-like object* pour *src* et *dst*.

`os.rmdir`(*path*, *, *dir_fd=None*)

Remove (delete) the directory *path*. If the directory does not exist or is not empty, a `FileNotFoundError` or an `OSError` is raised respectively. In order to remove whole directory trees, `shutil.rmtree()` can be used.

Cette fonction prend en charge *des chemins relatifs à des descripteurs de répertories*.

Lève un *événement d'audit* `os.rmdir` avec les arguments `path`, `dir_fd`.

Modifié dans la version 3.3 : Paramètre *dir_fd* ajouté.

Modifié dans la version 3.6 : Accepte un *path-like object*.

`os.scandir`(*path='.'*)

Return an iterator of `os.DirEntry` objects corresponding to the entries in the directory given by *path*. The entries are yielded in arbitrary order, and the special entries `'.'` and `'..'` are not included. If a file is removed from or added to the directory after creating the iterator, whether an entry for that file be included is unspecified.

Utiliser `scandir()` plutôt que `listdir()` peut significativement améliorer les performances des codes qui nécessitent aussi l'accès aux types des fichiers ou à leurs attributs, puisque les objets `os.DirEntry` exposent ces informations si le système d'exploitation les fournit en scannant le répertoire. Toutes les méthodes de `os.DirEntry` peuvent réaliser un appel système, mais `is_dir()` et `is_file()` n'en requièrent normalement un que pour les liens symboliques ; `os.DirEntry.stat()` nécessite toujours un appel système sous Unix, mais seulement pour les liens symboliques sous Windows.

path peut être un *path-like object*. Si *path* est de type `bytes` (directement ou indirectement à travers une interface `PathLike`), le type des attributs `name` et `path` de chaque `os.DirEntry` sera `bytes` ; dans toutes les autres circonstances, ils seront de type `str`.

Cette fonction peut également gérer *la spécification de descripteurs de fichiers*. Le descripteur doit référencer un répertoire.

Lève un *événement d'audit* `os.scandir` avec l'argument `path`.

L'itérateur `scandir()` gère le protocole *context manager* et possède la méthode suivante :

`scandir.close`()

Ferme l'itérateur et libère les ressources acquises.

Elle est appelée automatiquement quand l'itérateur est entièrement consommé ou collecté par le ramasse-miettes, ou quand une erreur survient durant l'itération. Il est cependant conseillé de l'appeler explicitement ou d'utiliser l'instruction `with`.

Ajouté dans la version 3.6.

L'exemple suivant montre une utilisation simple de `scandir()` pour afficher tous les fichiers (à part les répertoires) dans le chemin donné par *path* et ne débutant pas par `'.'`. L'appel `entry.is_file()` ne va généralement pas faire d'appel système supplémentaire :

```
with os.scandir(path) as it:
    for entry in it:
        if not entry.name.startswith('.') and entry.is_file():
            print(entry.name)
```

Note

On Unix-based systems, `scandir()` uses the system's opendir() and readdir() functions. On Windows, it uses the Win32 FindFirstFileW and FindNextFileW functions.

Ajouté dans la version 3.5.

Modifié dans la version 3.6 : Added support for the *context manager* protocol and the `close()` method. If a `scandir()` iterator is neither exhausted nor explicitly closed a `ResourceWarning` will be emitted in its destructor.

La fonction accepte un *path-like object*.

Modifié dans la version 3.7 : Ajout de la gestion des *descripteurs de fichiers* sur Unix.

class os.**DirEntry**

Objet donné par `scandir()` pour exposer le chemin du fichier et d'autres attributs de fichier d'une entrée du répertoire.

`scandir()` fournira autant d'informations que possible sans faire d'appels système additionnels. Quand un appel système `stat()` ou `lstat()` est réalisé, l'objet `os.DirEntry` mettra le résultat en cache.

Les instances `os.DirEntry` ne sont pas censées être stockées dans des structures de données à longue durée de vie ; si vous savez que les métadonnées du fichier ont changé ou si un certain temps s'est écoulé depuis l'appel à `scandir()`, appelez `os.stat(entry.path)` pour mettre à jour ces informations.

Puisque les méthodes de `os.DirEntry` peuvent réaliser des appels système, elles peuvent aussi lever des `OSError`. Si vous avez besoin d'un contrôle fin des erreurs, vous pouvez attraper les `OSError` en appelant les méthodes de `os.DirEntry` et les traiter comme il vous semble.

Pour être directement utilisable comme un *path-like object*, `os.DirEntry` implémente l'interface `PathLike`.

Les attributs et méthodes des instances de `os.DirEntry` sont les suivants :

name

Le nom de fichier de base de l'entrée, relatif à l'argument *path* de `scandir()`.

L'attribut `name` sera de type `bytes` si l'argument *path* de `scandir()` est de type `bytes`, sinon il sera de type `str`. Utilisez `fsdecode()` pour décoder des noms de fichiers de types *byte*.

path

Le nom entier de l'entrée : équivalent à `os.path.join(scandir_path, entry.name)` où *scandir_path* est l'argument *path* de `scandir()`. Le chemin est absolu uniquement si l'argument *path* de `scandir()` était absolu. Si l'argument *path* à la fonction `scandir()` est un *descripteur de fichier* l'attribut `path` sera égal à l'attribut `name`.

L'attribut `path` sera de type `bytes` si l'argument *path* de la fonction `scandir()` est de type `bytes`, sinon il sera de type `str`. Utilisez `fsdecode()` pour décoder des noms de fichiers de type *bytes*.

inode()

Renvoie le numéro d'*inode* de l'entrée.

Le résultat est mis en cache dans l'objet `os.DirEntry`. Utilisez `os.stat(entry.path, follow_symlinks=False).st_ino` pour obtenir l'information à jour.

Au premier appel non mis en cache, un appel système est requis sur Windows, mais pas sur Unix.

is_dir(*, *follow_symlinks=True*)

Renvoie `True` si cette entrée est un répertoire ou un lien symbolique pointant vers un répertoire ; renvoie `False` si l'entrée est (ou pointe vers) un autre type de fichier, ou s'il n'existe plus.

Si *follow_symlinks* vaut `False`, renvoie `True` uniquement si l'entrée est un répertoire (sans suivre les liens symboliques) ; renvoie `False` si l'entrée est n'importe quel autre type de fichier ou s'il n'existe plus.

Le résultat est mis en cache dans l'objet `os.DirEntry`, avec un cache séparé pour les valeurs `True` ou `False` de *follow_symlinks*. Appelez *os.stat()* avec *stat.S_ISDIR()* pour obtenir l'information à jour.

Au premier appel non mis en cache, aucun appel système n'est requis dans la plupart du temps. Spécifiquement, sans les liens symboliques, ni Windows, ni Unix ne requiert l'appel système, sauf sur certains systèmes de fichiers sur Unix, comme les système de fichiers de réseau qui renvoient `dirent.d_type == DT_UNKNOWN`. Si l'entrée est un lien symbolique, un appel système sera requis pour suivre le lien symbolique, à moins que *follow_symlinks* vaille `False`.

Cette méthode peut lever une *OSError* tout comme une *PermissionError*, mais *FileNotFoundError* est interceptée et pas levée.

is_file(*, *follow_symlinks=True*)

Renvoie `True` si l'entrée est un fichier ou un lien symbolique pointant vers un fichier, renvoie `False` si l'entrée pointe est (ou pointe sur) sur un dossier ou sur un répertoire ou autre entrée non-fichier, ou s'il n'existe plus.

Si *follow_symlinks* vaut `False`, renvoie `True` uniquement si cette entrée est un fichier (sans suivre les liens symboliques). Renvoie `False` si l'entrée est un répertoire ou une autre entrée non-fichier, ou s'il n'existe plus.

Le résultat est mis en cache dans l'objet `os.DirEntry`. La mise en cache, les appels système réalisés, et les exceptions levées sont les mêmes que pour *is_dir()*.

is_symlink()

Renvoie `True` si l'entrée est un lien symbolique (même cassé). Renvoie `False` si l'entrée pinte vers un répertoire ou tout autre type de fichier, ou s'il n'existe plus.

Le résultat est mis en cache dans l'objet `os.DirEntry`. Appelez *os.path.islink()* pour obtenir l'information à jour.

Au premier appel non mis en cache, aucun appel système n'est requis. Spécifiquement, ni Windows ni Unix ne requiert d'appel système, excepté sur certains systèmes de fichiers Unix qui renvoient `dirent.d_type == DT_UNKNOWN`.

Cette méthode peut lever une *OSError* tout comme une *PermissionError*, mais *FileNotFoundError* est interceptée et pas levée.

is_junction()

Return `True` if this entry is a junction (even if broken) ; return `False` if the entry points to a regular directory, any kind of file, a symlink, or if it doesn't exist anymore.

The result is cached on the `os.DirEntry` object. Call *os.path.isjunction()* to fetch up-to-date information.

Ajouté dans la version 3.12.

stat(*, *follow_symlinks=True*)

Renvoie un objet de type `stat.result` pour cette entrée. Cette méthode suit les liens symboliques par défaut. Pour avoir les statistiques sur un lien symbolique, ajouter l'argument `follow_symlinks=False`.

On Unix, this method always requires a system call. On Windows, it only requires a system call if *follow_symlinks* is `True` and the entry is a reparse point (for example, a symbolic link or directory junction).

Sur Windows, les attributs `st_ino`, `st_dev` et `st_nlink` de la classe `stat_result` sont toujours définis à 0. Appelez la fonction `os.stat()` pour avoir ces attributs.

Le résultat est mis en cache dans l'objet `os.DirEntry`, avec un cache séparé pour les valeurs `True` ou `False` de *follow_symlinks*. Appelez `os.stat()` pour obtenir l'information à jour.

Note that there is a nice correspondence between several attributes and methods of `os.DirEntry` and of `pathlib.Path`. In particular, the `name` attribute has the same meaning, as do the `is_dir()`, `is_file()`, `is_symlink()`, `is_junction()`, and `stat()` methods.

Ajouté dans la version 3.5.

Modifié dans la version 3.6 : Prise en charge de l'interface `PathLike`. Ajout du support des chemins `bytes` sous Windows.

Modifié dans la version 3.12 : The `st_ctime` attribute of a stat result is deprecated on Windows. The file creation time is properly available as `st_birthtime`, and in the future `st_ctime` may be changed to return zero or the metadata change time, if available.

`os.stat`(*path*, **, dir_fd=None, follow_symlinks=True*)

Récupère le statut d'un fichier ou d'un descripteur de fichier. Réalise l'équivalent d'un appel système `stat()` sur le chemin donné. *path* peut être exprimé comme une chaîne de caractères ou d'octets -- directement ou indirectement à travers une interface `PathLike` -- ou comme un descripteur de fichier ouvert. Renvoie un objet `stat_result`.

Cette fonction suit normalement les liens symboliques. Pour récupérer les informations d'un lien symbolique, ajoutez l'argument `follow_symlinks=False` ou utilisez la fonction `lstat()`.

Cette fonction peut supporter la *spécification d'un descripteur de fichier* et *le non-suivi des liens symboliques*.

On Windows, passing `follow_symlinks=False` will disable following all name-surrogate reparse points, which includes symlinks and directory junctions. Other types of reparse points that do not resemble links or that the operating system is unable to follow will be opened directly. When following a chain of multiple links, this may result in the original link being returned instead of the non-link that prevented full traversal. To obtain stat results for the final path in this case, use the `os.path.realpath()` function to resolve the path name as far as possible and call `lstat()` on the result. This does not apply to dangling symlinks or junction points, which will raise the usual exceptions.

Exemple :

```
>>> import os
>>> statinfo = os.stat('somefile.txt')
>>> statinfo
os.stat_result(st_mode=33188, st_ino=7876932, st_dev=234881026,
st_nlink=1, st_uid=501, st_gid=501, st_size=264, st_atime=1297230295,
st_mtime=1297230027, st_ctime=1297230027)
>>> statinfo.st_size
264
```

Voir aussi

les fonctions `fstat()` et `lstat()`.

Modifié dans la version 3.3 : Added the *dir_fd* and *follow_symlinks* parameters, specifying a file descriptor instead of a path.

Modifié dans la version 3.6 : Accepte un *path-like object*.

Modifié dans la version 3.8 : On Windows, all reparse points that can be resolved by the operating system are now followed, and passing `follow_symlinks=False` disables following all name surrogate reparse points.

If the operating system reaches a reparse point that it is not able to follow, *stat* now returns the information for the original path as if `follow_symlinks=False` had been specified instead of raising an error.

class `os.`**`stat_result`**

Object whose attributes correspond roughly to the members of the `stat` structure. It is used for the result of `os.stat()`, `os.fstat()` and `os.lstat()`.

Attributs :

`st_mode`

Mode du fichier : type du fichier et bits de mode du fichier (permissions).

`st_ino`

Dépendant de la plateforme, mais lorsqu'il ne vaut pas zéro il identifie de manière unique le fichier pour une certaine valeur de `st_dev`. Typiquement :
— le numéro d'*inode* sur Unix,
— l'index de fichier sur Windows

`st_dev`

Identifiant du périphérique sur lequel ce fichier se trouve.

`st_nlink`

Nombre de liens matériels.

`st_uid`

Identifiant d'utilisateur du propriétaire du fichier.

`st_gid`

Identifiant de groupe du propriétaire du fichier.

`st_size`

Taille du fichier en *bytes* si c'est un fichier normal ou un lien symbolique. La taille d'un lien symbolique est la longueur du nom de chemin qu'il contient sans le byte nul final.

Horodatages :

`st_atime`

Moment de l'accès le plus récent, exprimé en secondes.

`st_mtime`

Moment de la modification de contenu la plus récente, exprimé en secondes.

`st_ctime`

Time of most recent metadata change expressed in seconds.

Modifié dans la version 3.12 : `st_ctime` is deprecated on Windows. Use `st_birthtime` for the file creation time. In the future, `st_ctime` will contain the time of the most recent metadata change, as for other platforms.

`st_atime_ns`

Moment de l'accès le plus récent, exprimé en nanosecondes, par un entier.

Ajouté dans la version 3.3.

`st_mtime_ns`

Moment de la modification de contenu la plus récente, exprimé en nanosecondes, par un entier.

Ajouté dans la version 3.3.

`st_ctime_ns`

Time of most recent metadata change expressed in nanoseconds as an integer.

Ajouté dans la version 3.3.

Modifié dans la version 3.12 : `st_ctime_ns` is deprecated on Windows. Use `st_birthtime_ns` for the file creation time. In the future, `st_ctime` will contain the time of the most recent metadata change,

as for other platforms.

st_birthtime

Time of file creation expressed in seconds. This attribute is not always available, and may raise `AttributeError`.

Modifié dans la version 3.12 : `st_birthtime` is now available on Windows.

st_birthtime_ns

Time of file creation expressed in nanoseconds as an integer. This attribute is not always available, and may raise `AttributeError`.

Ajouté dans la version 3.12.

> **Note**
>
> The exact meaning and resolution of the `st_atime`, `st_mtime`, `st_ctime` and `st_birthtime` attributes depend on the operating system and the file system. For example, on Windows systems using the FAT32 file systems, `st_mtime` has 2-second resolution, and `st_atime` has only 1-day resolution. See your operating system documentation for details.
>
> Similarly, although `st_atime_ns`, `st_mtime_ns`, `st_ctime_ns` and `st_birthtime_ns` are always expressed in nanoseconds, many systems do not provide nanosecond precision. On systems that do provide nanosecond precision, the floating-point object used to store `st_atime`, `st_mtime`, `st_ctime` and `st_birthtime` cannot preserve all of it, and as such will be slightly inexact. If you need the exact timestamps you should always use `st_atime_ns`, `st_mtime_ns`, `st_ctime_ns` and `st_birthtime_ns`.

Sur certains systèmes Unix (tels que Linux), les attributs suivants peuvent également être disponibles :

st_blocks

Nombre de blocs de 512 *bytes* alloués pour le fichier. Cette valeur peut être inférieure à `st_size`/512 quand le fichier a des trous.

st_blksize

Taille de bloc « préférée » pour des entrées-sorties efficaces avec le système de fichiers. Écrire dans un fichier avec des blocs plus petits peut causer des modifications (lecture-écriture-réécriture) inefficaces.

st_rdev

Type de périphérique si l'*inode* représente un périphérique.

st_flags

Marqueurs définis par l'utilisateur pour le fichier.

Sur d'autres systèmes Unix (tels que FreeBSD), les attributs suivants peuvent être disponibles (mais peuvent être complétés uniquement lorsque le super-utilisateur *root* tente de les utiliser) :

st_gen

Nombre de génération de fichier.

Sur les systèmes Solaris et dérivés, les attributs suivants peuvent également être disponibles :

st_fstype

Chaîne qui identifie de manière unique le type du système de fichiers qui contient le fichier.

On macOS systems, the following attributes may also be available :

st_rsize

Taillé réelle du fichier.

st_creator

Créateur du fichier.

st_type

Type du fichier.

On Windows systems, the following attributes are also available :

st_file_attributes

Windows file attributes : `dwFileAttributes` member of the `BY_HANDLE_FILE_INFORMATION` structure returned by `GetFileInformationByHandle()`. See the `FILE_ATTRIBUTE_* <stat.FILE_ATTRIBUTE_ARCHIVE>` constants in the *stat* module.

Ajouté dans la version 3.5.

st_reparse_tag

When *st_file_attributes* has the *FILE_ATTRIBUTE_REPARSE_POINT* set, this field contains the tag identifying the type of reparse point. See the *IO_REPARSE_TAG_** constants in the *stat* module.

The standard module *stat* defines functions and constants that are useful for extracting information from a `stat` structure. (On Windows, some items are filled with dummy values.)

For backward compatibility, a *stat_result* instance is also accessible as a tuple of at least 10 integers giving the most important (and portable) members of the `stat` structure, in the order *st_mode*, *st_ino*, *st_dev*, *st_nlink*, *st_uid*, *st_gid*, *st_size*, *st_atime*, *st_mtime*, *st_ctime*. More items may be added at the end by some implementations. For compatibility with older Python versions, accessing *stat_result* as a tuple always returns integers.

Modifié dans la version 3.5 : Windows renvoie maintenant l'index du fichier dans l'attribut *st_ino*, lorsqu'il est disponible.

Modifié dans la version 3.7 : Ajout de l'attribut *st_fstype* sur Solaris et dérivés.

Modifié dans la version 3.8 : Added the *st_reparse_tag* member on Windows.

Modifié dans la version 3.8 : On Windows, the *st_mode* member now identifies special files as `S_IFCHR`, `S_IFIFO` or `S_IFBLK` as appropriate.

Modifié dans la version 3.12 : On Windows, *st_ctime* is deprecated. Eventually, it will contain the last metadata change time, for consistency with other platforms, but for now still contains creation time. Use *st_birthtime* for the creation time.

On Windows, *st_ino* may now be up to 128 bits, depending on the file system. Previously it would not be above 64 bits, and larger file identifiers would be arbitrarily packed.

On Windows, *st_rdev* no longer returns a value. Previously it would contain the same as *st_dev*, which was incorrect.

Added the *st_birthtime* member on Windows.

`os.statvfs(path)`

Perform a `statvfs()` system call on the given path. The return value is an object whose attributes describe the filesystem on the given path, and correspond to the members of the `statvfs` structure, namely : `f_bsize`, `f_frsize`, `f_blocks`, `f_bfree`, `f_bavail`, `f_files`, `f_ffree`, `f_favail`, `f_flag`, `f_namemax`, `f_fsid`.

Deux constantes de module sont définies pour le champ-de-bits de l'attribut `f_flag` : si `SR_RDONLY` est activé, le système de fichiers est monté en lecture-seule, et si `ST_NOSUID` est activé, la sémantique des bits de *setuid* / *getuid* est désactivée ou non gérée.

Des constantes de module supplémentaires sont définies pour les systèmes basés sur GNU/glibc. Ces constantes sont `ST_NODEV` (interdit l'accès aux fichiers spéciaux du périphérique), `ST_NOEXEC` (interdit l'exécution de programmes), `ST_SYNCHRONOUS` (les écritures sont synchronisées en une fois), `ST_MANDLOCK` (permet les verrous impératifs sur un système de fichiers), `ST_WRITE` (écrit sur les fichiers/répertoires/liens symboliques), `ST_APPEND` (fichiers en ajout-seul), `ST_IMMUTABLE` (fichiers immuables), `ST_NOATIME` (ne met pas à jour les moments d'accès), `ST_NODIRATIME` (ne met pas à jour les moments d'accès aux répertoires), `ST_REALTIME` (Met *atime* à jour relativement à *mtime* / *ctime*).

Cette fonction prend en charge *la spécification d'un descripteur de fichier*.

Availability : Unix.

Modifié dans la version 3.2 : Ajout des constantes `ST_RDONLY` et `ST_NOSUID`.

Modifié dans la version 3.3 : Ajout de la possibilité de spécifier *path* comme descripteur de fichier ouvert.

Modifié dans la version 3.4 : Ajout des constantes `ST_NODEV`, `ST_NOEXEC`, `ST_SYNCHRONOUS`, `ST_MANDLOCK`, `ST_WRITE`, `ST_APPEND`, `ST_IMMUTABLE`, `ST_NOATIME`, `ST_NODIRATIME`, et `ST_RELATIME`.

Modifié dans la version 3.6 : Accepte un *path-like object*.

Modifié dans la version 3.7 : Added the `f_fsid` attribute.

`os.`**`supports_dir_fd`**

A `set` object indicating which functions in the `os` module accept an open file descriptor for their *dir_fd* parameter. Different platforms provide different features, and the underlying functionality Python uses to implement the *dir_fd* parameter is not available on all platforms Python supports. For consistency's sake, functions that may support *dir_fd* always allow specifying the parameter, but will throw an exception if the functionality is used when it's not locally available. (Specifying `None` for *dir_fd* is always supported on all platforms.)

To check whether a particular function accepts an open file descriptor for its *dir_fd* parameter, use the `in` operator on `supports_dir_fd`. As an example, this expression evaluates to `True` if `os.stat()` accepts open file descriptors for *dir_fd* on the local platform :

```
os.stat in os.supports_dir_fd
```

Actuellement, le paramètre *dir_fd* ne fonctionne que sur les plate-formes Unix. Il ne fonctionne jamais sur Windows.

Ajouté dans la version 3.3.

`os.`**`supports_effective_ids`**

A `set` object indicating whether `os.access()` permits specifying `True` for its *effective_ids* parameter on the local platform. (Specifying `False` for *effective_ids* is always supported on all platforms.) If the local platform supports it, the collection will contain `os.access()` ; otherwise it will be empty.

This expression evaluates to `True` if `os.access()` supports `effective_ids=True` on the local platform :

```
os.access in os.supports_effective_ids
```

Currently *effective_ids* is only supported on Unix platforms ; it does not work on Windows.

Ajouté dans la version 3.3.

`os.`**`supports_fd`**

A `set` object indicating which functions in the `os` module permit specifying their *path* parameter as an open file descriptor on the local platform. Different platforms provide different features, and the underlying functionality Python uses to accept open file descriptors as *path* arguments is not available on all platforms Python supports.

To determine whether a particular function permits specifying an open file descriptor for its *path* parameter, use the `in` operator on `supports_fd`. As an example, this expression evaluates to `True` if `os.chdir()` accepts open file descriptors for *path* on your local platform :

```
os.chdir in os.supports_fd
```

Ajouté dans la version 3.3.

`os.`**`supports_follow_symlinks`**

A `set` object indicating which functions in the `os` module accept `False` for their *follow_symlinks* parameter on the local platform. Different platforms provide different features, and the underlying functionality Python uses to implement *follow_symlinks* is not available on all platforms Python supports. For consistency's sake, functions that may support *follow_symlinks* always allow specifying the parameter, but will throw an exception if the functionality is used when it's not locally available. (Specifying `True` for *follow_symlinks* is always supported on all platforms.)

To check whether a particular function accepts `False` for its *follow_symlinks* parameter, use the `in` operator on `supports_follow_symlinks`. As an example, this expression evaluates to `True` if you may specify `follow_symlinks=False` when calling `os.stat()` on the local platform :

```
os.stat in os.supports_follow_symlinks
```

Ajouté dans la version 3.3.

os.**symlink**(*src*, *dst*, *target_is_directory=False*, ***, *dir_fd=None*)

Crée un lien symbolique pointant vers *src* et appelé *dst*.

Sur Windows, un lien symbolique représente soit lien vers un fichier, soit lien vers un répertoire mais ne s'adapte pas dynamiquement au type de la cible. Si la cible existe le lien sera crée du même type que sa cible. Dans le cas où cible n'existe pas, si *target_is_directory* vaut `True` le lien symbolique sera créé comme un répertoire, sinon comme un fichier (par défaut). Sur les autres plateformes, *target_id_directory* est ignoré.

Cette fonction prend en charge *des chemins relatifs à des descripteurs de répertories*.

> **Note**
>
> On newer versions of Windows 10, unprivileged accounts can create symlinks if Developer Mode is enabled. When Developer Mode is not available/enabled, the *SeCreateSymbolicLinkPrivilege* privilege is required, or the process must be run as an administrator.
>
> `OSError` est levée quand la fonction est appelée par un utilisateur sans privilèges.

Lève un *événement d'audit* `os.symlink` avec les arguments `src`, `dst`, `dir_fd`.

Availability : Unix, Windows.

The function is limited on WASI, see *Plateformes WebAssembly* for more information.

Modifié dans la version 3.2 : Prise en charge des les liens symboliques sur Windows 6.0 (Vista).

Modifié dans la version 3.3 : Added the *dir_fd* parameter, and now allow *target_is_directory* on non-Windows platforms.

Modifié dans la version 3.6 : Accepte un *path-like object* pour *src* et *dst*.

Modifié dans la version 3.8 : Added support for unelevated symlinks on Windows with Developer Mode.

os.**sync**()

Force l'écriture de tout sur le disque.

Availability : Unix.

Ajouté dans la version 3.3.

os.**truncate**(*path*, *length*)

Tronque le fichier correspondant à *path*, afin qu'il soit au maximum long de *length* bytes.

Cette fonction prend en charge *la spécification d'un descripteur de fichier*.

Lève un *événement d'audit* `os.truncate` avec les arguments `path`, `length`.

Availability : Unix, Windows.

Ajouté dans la version 3.3.

Modifié dans la version 3.5 : Prise en charge de Windows

Modifié dans la version 3.6 : Accepte un *path-like object*.

os.**unlink**(*path*, ***, *dir_fd=None*)

Supprime (retire) le fichier *path*. Cette fonction est sémantiquement identique à `remove()`. Le nom `unlink` est un nom Unix traditionnel. Veuillez voir la documentation de `remove()` pour plus d'informations.

Lève un *événement d'audit* `os.remove` avec les arguments `path`, `dir_fd`.

Modifié dans la version 3.3 : Paramètre *dir_fd* ajouté.

Modifié dans la version 3.6 : Accepte un *path-like object*.

os.**utime**(*path*, *times=None*, *, [*ns*,]*dir_fd=None*, *follow_symlinks=True*)

Définit les derniers moments d'accès et de modification du fichier spécifiés par *path*.

La fonction `utime()` prend deux paramètres optionnels, *times* et *ns*. Ils spécifient le temps mis pour *path* et est utilisé comme suit :

— Si *ns* est spécifié, ce doit être un couple de la forme `(atime_ns, mtime_ns)` où chaque membre est un entier qui exprime des nanosecondes.
— Si *times* ne vaut pas `None`, ce doit être un couple de la forme `(atime, mtime)` où chaque membre est un entier ou une expression à virgule flottante.
— Si *times* vaut `None`, et *ns* est non-spécifié. C'est équivalent à spécifier `ns = (atime_ns, mtime_ns)` où les deux moments sont le moment actuel.

Il est erroné de spécifier des *n*-uplets pour *times* et *ns* à la fois.

Note that the exact times you set here may not be returned by a subsequent `stat()` call, depending on the resolution with which your operating system records access and modification times ; see `stat()`. The best way to preserve exact times is to use the *st_atime_ns* and *st_mtime_ns* fields from the `os.stat()` result object with the *ns* parameter to `utime()`.

Cette fonction prend en charge *la spécification d'un descripteur de fichier*, *les chemins relatifs à des descripteurs de répertoires*, et *le non-suivi des liens symboliques*.

Lève un *événement d'audit* `os.utime` avec les arguments `path`, `times`, `ns`, `dir_fd`.

Modifié dans la version 3.3 : Ajoute la prise en charge d'un descripteur de fichier pour *path* et des paramètres *dir_fd*, *follow_symlinks* et *ns*.

Modifié dans la version 3.6 : Accepte un *path-like object*.

os.**walk**(*top*, *topdown=True*, *onerror=None*, *followlinks=False*)

Génère les noms de fichier dans un arbre de répertoire en parcourant l'arbre soit de bas-en-haut, soit de haut-en-bas. Pour chaque répertoire dans l'arbre enraciné en le répertoire *rop* (incluant ledit répertoire *top*), fournit un triplet `(dirpath, dirnames, filenames)`.

dirpath is a string, the path to the directory. *dirnames* is a list of the names of the subdirectories in *dirpath* (including symlinks to directories, and excluding `'.'` and `'..'`). *filenames* is a list of the names of the non-directory files in *dirpath*. Note that the names in the lists contain no path components. To get a full path (which begins with *top*) to a file or directory in *dirpath*, do `os.path.join(dirpath, name)`. Whether or not the lists are sorted depends on the file system. If a file is removed from or added to the *dirpath* directory during generating the lists, whether a name for that file be included is unspecified.

Si l'argument optionnel *topdown* vaut `True` ou n'est pas spécifié, le triplet pour un répertoire est généré avant les triplets de tous ses sous-répertoires (les répertoires sont générés de haut-en-bas). Si *topdown* vaut `False`, le triplet pour un répertoire est généré après les triplets de tous ses sous-répertoires (les répertoires sont générés de bas-en-haut). Peu importe la valeur de *topdown*, la liste des sous-répertoires est récupérée avant que les *n*-uplets pour le répertoires et ses sous-répertoires ne soient générés.

Quand *topdown* vaut `True`, l'appelant peut modifier la liste *dirnames* en place (par exemple en utilisant `del` ou l'assignation par *slicing* (par tranches)) et `walk()` ne fera sa récursion que dans les sous-répertoires dont le nom reste dans *dirnames* ; cela peut être utilisé pour élaguer la recherche, imposer un ordre particulier de visite, ou encore pour informer `walk()` des répertoires créés par l'appelant ou renommés avant qu'il quitte `walk()` à nouveau. Modifier *dirnames* quand *topdown* vaut `False` n'a aucun effet dur le comportement du parcours parce qu'en mode bas-en-haut, les répertoires dans *dirnames* sont générés avant que *dirpath* ne soit lui-même généré.

Par défaut, les erreurs d'un appel à `scandir()` sont ignorées. Si l'argument optionnel *onerror* est spécifié, il doit être une fonction qui sera appelée avec un seul argument, une instance de `OSError`. Elle peut rapporter l'erreur et continuer le parcours, ou lever l'exception pour avorter le parcours. Notez que le nom de fichier est disponible dans l'attribut `filename` de l'objet exception.

Par défaut, `walk()` ne parcourra pas les liens symboliques qui mènent à un répertoire. Définissez *followlinks* avec `True` pour visiter les répertoires pointés par des liens symboliques sur les systèmes qui le gère.

> **Note**
>
> Soyez au courant que définir *followlinks* avec `True` peut mener à une récursion infinie si un lien pointe vers un répertoire parent de lui-même. `walk()` ne garde pas de trace des répertoires qu'il a déjà visité.

> **Note**
>
> Si vous passez un chemin relatif, ne changer pas le répertoire de travail actuel entre deux exécutions de `walk()`. `walk()` ne change jamais le répertoire actuel, et suppose que l'appelant ne le fait pas non plus.

Cet exemple affiche le nombre de bytes pris par des fichiers non-répertoires dans chaque répertoire à partir du répertoire de départ, si ce n'est qu'il ne cherche pas après un sous-répertoire CSV :

```
import os
from os.path import join, getsize
for root, dirs, files in os.walk('python/Lib/email'):
    print(root, "consumes", end=" ")
    print(sum(getsize(join(root, name)) for name in files), end=" ")
    print("bytes in", len(files), "non-directory files")
    if 'CVS' in dirs:
        dirs.remove('CVS')  # don't visit CVS directories
```

Dans l'exemple suivant (simple implémentation d'un `shutil.rmtree()`), parcourir l'arbre de bas-en-haut est essentiel : `rmdir()` ne permet pas de supprimer un répertoire avant qu'un ne soit vide :

```
# Delete everything reachable from the directory named in "top",
# assuming there are no symbolic links.
# CAUTION:  This is dangerous!  For example, if top == '/', it
# could delete all your disk files.
import os
for root, dirs, files in os.walk(top, topdown=False):
    for name in files:
        os.remove(os.path.join(root, name))
    for name in dirs:
        os.rmdir(os.path.join(root, name))
os.rmdir(top)
```

Lève un *événement d'audit* `os.walk` avec les arguments `top`, `topdown`, `onerror`, `followlinks`.

Modifié dans la version 3.5 : Cette fonction appelle maintenant `os.scandir()` au lieu de `os.listdir()`, ce qui la rend plus rapide en réduisant le nombre d'appels à `os.stat()`.

Modifié dans la version 3.6 : Accepte un *path-like object*.

`os.`**`fwalk`**(*top='.'*, *topdown=True*, *onerror=None*, ***, *follow_symlinks=False*, *dir_fd=None*)

Se comporte exactement comme `walk()`, si ce n'est qu'il fournit un quadruplet `(dirpath, dirnames, filenames, dirfd)`, et gère `dir_fd`.

dirpath, *dirnames* et *filenames* sont identiques à la sortie de `walk()` et *dirfd* est un descripteur de fichier faisant référence au répertoire *dirpath*.

Cette fonction prend toujours en charge *les chemins relatifs à des descripteurs de fichiers* et *le non-suivi des liens symboliques*. Notez cependant qu'à l'inverse des autres fonctions, la valeur par défaut de *follow_symlinks* pour `walk()` est `False`.

> **Note**

> Puisque `fwalk()` fournit des descripteurs de fichiers, ils ne sont valides que jusque la prochaine itération. Donc vous devriez les dupliquer (par exemple avec `dup()`) si vous désirez les garder plus longtemps.

Cet exemple affiche le nombre de bytes pris par des fichiers non-répertoires dans chaque répertoire à partir du répertoire de départ, si ce n'est qu'il ne cherche pas après un sous-répertoire CSV :

```
import os
for root, dirs, files, rootfd in os.fwalk('python/Lib/email'):
    print(root, "consumes", end="")
    print(sum([os.stat(name, dir_fd=rootfd).st_size for name in files]),
          end="")
    print("bytes in", len(files), "non-directory files")
    if 'CVS' in dirs:
        dirs.remove('CVS')  # don't visit CVS directories
```

Dans le prochain exemple, parcourir l'arbre de bas-en-haut est essentiel : `rmdir()` ne permet pas de supprimer un répertoire avant qu'il ne soit vide :

```
# Delete everything reachable from the directory named in "top",
# assuming there are no symbolic links.
# CAUTION:  This is dangerous!  For example, if top == '/', it
# could delete all your disk files.
import os
for root, dirs, files, rootfd in os.fwalk(top, topdown=False):
    for name in files:
        os.unlink(name, dir_fd=rootfd)
    for name in dirs:
        os.rmdir(name, dir_fd=rootfd)
```

Lève un *événement d'audit* `os.fwalk` avec les arguments `top`, `topdown`, `onerror`, `follow_symlinks`, `dir_fd`.

Availability : Unix.

Ajouté dans la version 3.3.

Modifié dans la version 3.6 : Accepte un *path-like object*.

Modifié dans la version 3.7 : Ajout de la gestion des chemins de type `bytes`.

`os.`**`memfd_create`**(*name*[, *flags=os.MFD_CLOEXEC*])

Create an anonymous file and return a file descriptor that refers to it. *flags* must be one of the `os.MFD_*` constants available on the system (or a bitwise ORed combination of them). By default, the new file descriptor is *non-inheritable*.

The name supplied in *name* is used as a filename and will be displayed as the target of the corresponding symbolic link in the directory `/proc/self/fd/`. The displayed name is always prefixed with `memfd:` and serves only for debugging purposes. Names do not affect the behavior of the file descriptor, and as such multiple files can have the same name without any side effects.

Availability : Linux >= 3.17 with glibc >= 2.27.

Ajouté dans la version 3.8.

`os.`**`MFD_CLOEXEC`**

`os.`**`MFD_ALLOW_SEALING`**

`os.`**`MFD_HUGETLB`**

`os.`**`MFD_HUGE_SHIFT`**

`os.`**`MFD_HUGE_MASK`**

`os.`**`MFD_HUGE_64KB`**

`os.`**`MFD_HUGE_512KB`**

os.**MFD_HUGE_1MB**

os.**MFD_HUGE_2MB**

os.**MFD_HUGE_8MB**

os.**MFD_HUGE_16MB**

os.**MFD_HUGE_32MB**

os.**MFD_HUGE_256MB**

os.**MFD_HUGE_512MB**

os.**MFD_HUGE_1GB**

os.**MFD_HUGE_2GB**

os.**MFD_HUGE_16GB**

These flags can be passed to `memfd_create()`.

Availability : Linux >= 3.17 with glibc >= 2.27

The `MFD_HUGE*` flags are only available since Linux 4.14.

Ajouté dans la version 3.8.

os.**eventfd**(*initval*[, *flags=os.EFD_CLOEXEC*])

Create and return an event file descriptor. The file descriptors supports raw `read()` and `write()` with a buffer size of 8, `select()`, `poll()` and similar. See man page `eventfd(2)` for more information. By default, the new file descriptor is *non-inheritable*.

initval is the initial value of the event counter. The initial value must be a 32 bit unsigned integer. Please note that the initial value is limited to a 32 bit unsigned int although the event counter is an unsigned 64 bit integer with a maximum value of 2^{64}-2.

flags can be constructed from `EFD_CLOEXEC`, `EFD_NONBLOCK`, and `EFD_SEMAPHORE`.

If `EFD_SEMAPHORE` is specified and the event counter is non-zero, `eventfd_read()` returns 1 and decrements the counter by one.

If `EFD_SEMAPHORE` is not specified and the event counter is non-zero, `eventfd_read()` returns the current event counter value and resets the counter to zero.

If the event counter is zero and `EFD_NONBLOCK` is not specified, `eventfd_read()` blocks.

`eventfd_write()` increments the event counter. Write blocks if the write operation would increment the counter to a value larger than 2^{64}-2.

Exemple :

```
import os

# semaphore with start value '1'
fd = os.eventfd(1, os.EFD_SEMAPHORE | os.EFC_CLOEXEC)
try:
    # acquire semaphore
    v = os.eventfd_read(fd)
    try:
        do_work()
    finally:
        # release semaphore
        os.eventfd_write(fd, v)
finally:
    os.close(fd)
```

Availability : Linux >= 2.6.27 with glibc >= 2.8

Ajouté dans la version 3.10.

os.**eventfd_read**(*fd*)

Read value from an `eventfd()` file descriptor and return a 64 bit unsigned int. The function does not verify that *fd* is an `eventfd()`.

Availability : Linux >= 2.6.27

Ajouté dans la version 3.10.

os.**eventfd_write**(*fd*, *value*)

Add value to an `eventfd()` file descriptor. *value* must be a 64 bit unsigned int. The function does not verify that *fd* is an `eventfd()`.

Availability : Linux >= 2.6.27

Ajouté dans la version 3.10.

os.**EFD_CLOEXEC**

Set close-on-exec flag for new `eventfd()` file descriptor.

Availability : Linux >= 2.6.27

Ajouté dans la version 3.10.

os.**EFD_NONBLOCK**

Set `O_NONBLOCK` status flag for new `eventfd()` file descriptor.

Availability : Linux >= 2.6.27

Ajouté dans la version 3.10.

os.**EFD_SEMAPHORE**

Provide semaphore-like semantics for reads from an `eventfd()` file descriptor. On read the internal counter is decremented by one.

Availability : Linux >= 2.6.30

Ajouté dans la version 3.10.

Timer File Descriptors

Ajouté dans la version 3.13.

These functions provide support for Linux's *timer file descriptor* API. Naturally, they are all only available on Linux.

os.**timerfd_create**(*clockid*, /, *, *flags=0*)

Create and return a timer file descriptor (*timerfd*).

The file descriptor returned by `timerfd_create()` supports :
— `read()`
— `select()`
— `poll()`
The file descriptor's `read()` method can be called with a buffer size of 8. If the timer has already expired one or more times, `read()` returns the number of expirations with the host's endianness, which may be converted to an `int` by `int.from_bytes(x, byteorder=sys.byteorder)`.

`select()` and `poll()` can be used to wait until timer expires and the file descriptor is readable.

clockid must be a valid *clock ID*, as defined in the `time` module :
— `time.CLOCK_REALTIME`
— `time.CLOCK_MONOTONIC`
— `time.CLOCK_BOOTTIME` (Since Linux 3.15 for timerfd_create)
If *clockid* is `time.CLOCK_REALTIME`, a settable system-wide real-time clock is used. If system clock is changed, timer setting need to be updated. To cancel timer when system clock is changed, see `TFD_TIMER_CANCEL_ON_SET`.

If *clockid* is `time.CLOCK_MONOTONIC`, a non-settable monotonically increasing clock is used. Even if the system clock is changed, the timer setting will not be affected.

If *clockid* is `time.CLOCK_BOOTTIME`, same as `time.CLOCK_MONOTONIC` except it includes any time that the system is suspended.

The file descriptor's behaviour can be modified by specifying a *flags* value. Any of the following variables may used, combined using bitwise OR (the `|` operator) :
— `TFD_NONBLOCK`
— `TFD_CLOEXEC`
If `TFD_NONBLOCK` is not set as a flag, `read()` blocks until the timer expires. If it is set as a flag, `read()` doesn't block, but If there hasn't been an expiration since the last call to read, `read()` raises `OSError` with `errno` is set to `errno.EAGAIN`.

`TFD_CLOEXEC` is always set by Python automatically.

The file descriptor must be closed with `os.close()` when it is no longer needed, or else the file descriptor will be leaked.

Voir aussi

The `timerfd_create(2)` man page.

Availability : Linux >= 2.6.27 with glibc >= 2.8

Ajouté dans la version 3.13.

`os.`**`timerfd_settime`**(*fd*, /, *, *flags=flags*, *initial=0.0*, *interval=0.0*)

Alter a timer file descriptor's internal timer. This function operates the same interval timer as `timerfd_settime_ns()`.

fd must be a valid timer file descriptor.

The timer's behaviour can be modified by specifying a *flags* value. Any of the following variables may used, combined using bitwise OR (the `|` operator) :
— `TFD_TIMER_ABSTIME`
— `TFD_TIMER_CANCEL_ON_SET`
The timer is disabled by setting *initial* to zero (`0`). If *initial* is equal to or greater than zero, the timer is enabled. If *initial* is less than zero, it raises an `OSError` exception with `errno` set to `errno.EINVAL`

By default the timer will fire when *initial* seconds have elapsed. (If *initial* is zero, timer will fire immediately.)

However, if the `TFD_TIMER_ABSTIME` flag is set, the timer will fire when the timer's clock (set by *clockid* in `timerfd_create()`) reaches *initial* seconds.

The timer's interval is set by the *interval* `float`. If *interval* is zero, the timer only fires once, on the initial expiration. If *interval* is greater than zero, the timer fires every time *interval* seconds have elapsed since the previous expiration. If *interval* is less than zero, it raises `OSError` with `errno` set to `errno.EINVAL`

If the `TFD_TIMER_CANCEL_ON_SET` flag is set along with `TFD_TIMER_ABSTIME` and the clock for this timer is `time.CLOCK_REALTIME`, the timer is marked as cancelable if the real-time clock is changed discontinuously. Reading the descriptor is aborted with the error ECANCELED.

Linux manages system clock as UTC. A daylight-savings time transition is done by changing time offset only and doesn't cause discontinuous system clock change.

Discontinuous system clock change will be caused by the following events :
— `settimeofday`
— `clock_settime`
— set the system date and time by `date` command
Return a two-item tuple of (`next_expiration`, `interval`) from the previous timer state, before this function executed.

Voir aussi

`timerfd_create(2)`, `timerfd_settime(2)`, `settimeofday(2)`, `clock_settime(2)`, and

> *date(1)*.

Availability : Linux >= 2.6.27 with glibc >= 2.8

Ajouté dans la version 3.13.

os.**timerfd_settime_ns**(*fd*, /, *, *flags=0*, *initial=0*, *interval=0*)

Similar to *timerfd_settime()*, but use time as nanoseconds. This function operates the same interval timer as *timerfd_settime()*.

Availability : Linux >= 2.6.27 with glibc >= 2.8

Ajouté dans la version 3.13.

os.**timerfd_gettime**(*fd*, /)

Return a two-item tuple of floats (`next_expiration`, `interval`).

`next_expiration` denotes the relative time until next the timer next fires, regardless of if the *TFD_TIMER_ABSTIME* flag is set.

`interval` denotes the timer's interval. If zero, the timer will only fire once, after `next_expiration` seconds have elapsed.

> **Voir aussi**
>
> *timerfd_gettime(2)*

Availability : Linux >= 2.6.27 with glibc >= 2.8

Ajouté dans la version 3.13.

os.**timerfd_gettime_ns**(*fd*, /)

Similar to *timerfd_gettime()*, but return time as nanoseconds.

Availability : Linux >= 2.6.27 with glibc >= 2.8

Ajouté dans la version 3.13.

os.**TFD_NONBLOCK**

A flag for the *timerfd_create()* function, which sets the *O_NONBLOCK* status flag for the new timer file descriptor. If *TFD_NONBLOCK* is not set as a flag, *read()* blocks.

Availability : Linux >= 2.6.27 with glibc >= 2.8

Ajouté dans la version 3.13.

os.**TFD_CLOEXEC**

A flag for the *timerfd_create()* function, If *TFD_CLOEXEC* is set as a flag, set close-on-exec flag for new file descriptor.

Availability : Linux >= 2.6.27 with glibc >= 2.8

Ajouté dans la version 3.13.

os.**TFD_TIMER_ABSTIME**

A flag for the *timerfd_settime()* and *timerfd_settime_ns()* functions. If this flag is set, *initial* is interpreted as an absolute value on the timer's clock (in UTC seconds or nanoseconds since the Unix Epoch).

Availability : Linux >= 2.6.27 with glibc >= 2.8

Ajouté dans la version 3.13.

os.**TFD_TIMER_CANCEL_ON_SET**

A flag for the *timerfd_settime()* and *timerfd_settime_ns()* functions along with

`TFD_TIMER_ABSTIME`. The timer is cancelled when the time of the underlying clock changes discontinuously.

Availability : Linux >= 2.6.27 with glibc >= 2.8

Ajouté dans la version 3.13.

Attributs étendus pour Linux

Ajouté dans la version 3.3.

Toutes ces fonctions ne sont disponibles que sur Linux.

`os.`**`getxattr`**(*path*, *attribute*, **, *follow_symlinks=True*)

Renvoie la valeur de l'attribut étendu *attribute* du système de fichiers pour le chemin *path*. *attribute* peut être une chaîne de caractères ou d'octets (directement ou indirectement à travers une interface `PathLike`). Si c'est une chaîne de caractères, elle est encodée avec l'encodage du système de fichiers.

Cette fonction peut supporter la *spécification d'un descripteur de fichier* et *le non-suivi des liens symboliques*.

Lève un *événement d'audit* `os.getxattr` avec les arguments `path`, `attribute`.

Modifié dans la version 3.6 : Accepte un *path-like object* pour *path* et *attribute*.

`os.`**`listxattr`**(*path=None*, **, *follow_symlinks=True*)

Renvoie une liste d'attributs du système de fichiers étendu pour *path*. Les attributs dans le liste sont représentés par des chaînes de caractères et sont décodés avec l'encodage du système de fichier. Si *path* vaut `None`, `listxattr()` examinera le répertoire actuel.

Cette fonction peut supporter la *spécification d'un descripteur de fichier* et *le non-suivi des liens symboliques*.

Lève un *événement d'audit* `os.listxattr` avec l'argument `path`.

Modifié dans la version 3.6 : Accepte un *path-like object*.

`os.`**`removexattr`**(*path*, *attribute*, **, *follow_symlinks=True*)

Removes the extended filesystem attribute *attribute* from *path*. *attribute* should be bytes or str (directly or indirectly through the `PathLike` interface). If it is a string, it is encoded with the *filesystem encoding and error handler*.

Cette fonction peut supporter la *spécification d'un descripteur de fichier* et *le non-suivi des liens symboliques*.

Lève un *événement d'audit* `os.removexattr` avec les arguments `path`, `attribute`.

Modifié dans la version 3.6 : Accepte un *path-like object* pour *path* et *attribute*.

`os.`**`setxattr`**(*path*, *attribute*, *value*, *flags=0*, **, *follow_symlinks=True*)

Set the extended filesystem attribute *attribute* on *path* to *value*. *attribute* must be a bytes or str with no embedded NULs (directly or indirectly through the `PathLike` interface). If it is a str, it is encoded with the *filesystem encoding and error handler*. *flags* may be `XATTR_REPLACE` or `XATTR_CREATE`. If `XATTR_REPLACE` is given and the attribute does not exist, `ENODATA` will be raised. If `XATTR_CREATE` is given and the attribute already exists, the attribute will not be created and `EEXISTS` will be raised.

Cette fonction peut supporter la *spécification d'un descripteur de fichier* et *le non-suivi des liens symboliques*.

> **Note**
>
> Un bogue des versions inférieures à 2.6.39 du noyau Linux faisait que les marqueurs de *flags* étaient ignorés sur certains systèmes.

Lève un *événement d'audit* `os.setxattr` avec les arguments `path`, `attribute`, `value`, `flags`.

Modifié dans la version 3.6 : Accepte un *path-like object* pour *path* et *attribute*.

`os.`**`XATTR_SIZE_MAX`**

La taille maximum que peut faire la valeur d'un attribut étendu. Actuellement, c'est 64 KiB sur Lniux.

os.**XATTR_CREATE**

C'est une valeur possible pour l'argument *flags* de `setxattr()`. Elle indique que l'opération doit créer un attribut.

os.**XATTR_REPLACE**

C'est une valeur possible pour l'argument *flags* de `setxattr()`.Elle indique que l'opération doit remplacer un attribut existant.

16.1.7 Gestion des processus

Ces fonctions peuvent être utilisées pour créer et gérer des processus.

Les variantes des fonctions `exec*` prennent une liste d'arguments pour le nouveau programme chargé dans le processus. Dans tous les cas, le premier de ces arguments est passé au nouveau programme comme son propre nom plutôt que comme un argument qu'un utilisateur peut avoir tapé en ligne de commande. Pour les développeurs C, c'est l'argument `argv[0]` qui est passé à la fonction `main()` du programme. Par exemple, `os.execv('/bin/echo/', ['foo', 'bar'])` affichera uniquement `bar` sur la sortie standard ; `foo` semblera être ignoré.

os.**abort**()

Génère un signal `SIGABRT` au processus actuel. Sur Linux, le comportement par défaut est de produire un vidage système (*Core Dump*) ; sur Windows, le processus renvoie immédiatement un code d'erreur `3`. Attention : appeler cette fonction n'appellera pas le gestionnaire de signal Python enregistré par `SIGABRT` à l'aide de `signal.signal()`.

os.**add_dll_directory**(*path*)

Ajoute un chemin aux chemins de recherche de DLL.

This search path is used when resolving dependencies for imported extension modules (the module itself is resolved through `sys.path`), and also by `ctypes`.

Remove the directory by calling **close()** on the returned object or using it in a `with` statement.

See the Microsoft documentation for more information about how DLLs are loaded.

Lève un *événement d'audit* `os.add_dll_directory` avec l'argument `path`.

Availability : Windows.

Ajouté dans la version 3.8 : Previous versions of CPython would resolve DLLs using the default behavior for the current process. This led to inconsistencies, such as only sometimes searching `PATH` or the current working directory, and OS functions such as `AddDllDirectory` having no effect.

In 3.8, the two primary ways DLLs are loaded now explicitly override the process-wide behavior to ensure consistency. See the porting notes for information on updating libraries.

os.**execl**(*path*, *arg0*, *arg1*, ...)

os.**execle**(*path*, *arg0*, *arg1*, ..., *env*)

os.**execlp**(*file*, *arg0*, *arg1*, ...)

os.**execlpe**(*file*, *arg0*, *arg1*, ..., *env*)

os.**execv**(*path*, *args*)

os.**execve**(*path*, *args*, *env*)

os.**execvp**(*file*, *args*)

os.**execvpe**(*file*, *args*, *env*)

Ces fonctions exécutent toutes un nouveau programme, remplaçant le processus actuel, elles ne renvoient pas. Sur Unix, le nouvel exécutable est chargé dans le processus actuel, et aura le même identifiant de processus (PID) que l'appelant. Les erreurs seront reportées par des exceptions `OSError`.

Le processus actuel est remplacé immédiatement. Les fichiers objets et descripteurs de fichiers ne sont pas purgés, donc s'il est possible que des données aient été mises en tampon pour ces fichiers, vous devriez les purger manuellement en utilisant `sys.stdout.flush()` ou `os.fsync()` avant d'appeler une fonction `exec*`.

The "l" and "v" variants of the `exec*` functions differ in how command-line arguments are passed. The "l" variants are perhaps the easiest to work with if the number of parameters is fixed when the code is written ;

the individual parameters simply become additional parameters to the `execl*()` functions. The "v" variants are good when the number of parameters is variable, with the arguments being passed in a list or tuple as the *args* parameter. In either case, the arguments to the child process should start with the name of the command being run, but this is not enforced.

The variants which include a "p" near the end (`execlp()`, `execlpe()`, `execvp()`, and `execvpe()`) will use the `PATH` environment variable to locate the program *file*. When the environment is being replaced (using one of the `exec*e` variants, discussed in the next paragraph), the new environment is used as the source of the `PATH` variable. The other variants, `execl()`, `execle()`, `execv()`, and `execve()`, will not use the `PATH` variable to locate the executable ; *path* must contain an appropriate absolute or relative path. Relative paths must include at least one slash, even on Windows, as plain names will not be resolved.

Pour les fonctions `execle()`, `execlpe()`, `execve()`, et `execvpe()` (notez qu'elle finissent toutes par « e »), le paramètre *env* doit être un *mapping* qui est utilisé pour définir les variables d'environnement du nouveau processus (celles-ci sont utilisées à la place de l'environnement du nouveau processus). Les fonctions `execl()`, `execlp()`, `execv()`, et `execvp()` causent toutes un héritage de l'environnement du processus actuel par le processus fils.

Pour `execve()`, sur certaines plate-formes, *path* peut également être spécifié par un descripteur de fichier ouvert. Cette fonctionnalité peut ne pas être gérée sur votre plate-forme. Vous pouvez vérifier si c'est disponible en utilisant `os._supports_fd`. Si c'est indisponible, l'utiliser lèvera une `NotImplementedError`.

Lève un *événement d'audit* `os.exec` avec les arguments `path`, `args`, `env`.

Availability : Unix, Windows, not WASI, not Android, not iOS.

Modifié dans la version 3.3 : Added support for specifying *path* as an open file descriptor for `execve()`.

Modifié dans la version 3.6 : Accepte un *path-like object*.

`os.`**`_exit`**`(n)`

Quitte le processus avec le statut *n*, sans appeler les gestionnaires de nettoyage, sans purger les tampons des fichiers, etc.

> **Note**
>
> The standard way to exit is `sys.exit(n)`. `_exit()` should normally only be used in the child process after a `fork()`.

Les codes de sortie suivants sont définis et peuvent être utilisés avec `_exit()`, mais ils ne sont pas nécessaires. Ils sont typiquement utilisés pour les programmes systèmes écrits en Python, comme un programme de gestion de l'exécution des commandes d'un serveur de mails.

> **Note**
>
> Certaines de ces valeurs peuvent ne pas être disponibles sur toutes les plate-formes Unix étant donné qu'il en existe des variations. Ces constantes sont définies là où elles sont définies par la plate-forme sous-jacente.

`os.`**`EX_OK`**

Exit code that means no error occurred. May be taken from the defined value of `EXIT_SUCCESS` on some platforms. Generally has a value of zero.

Availability : Unix, Windows.

`os.`**`EX_USAGE`**

Code de sortie signifiant que les commandes n'ont pas été utilisées correctement, comme quand le mauvais nombre d'arguments a été donné.

Availability : Unix, not WASI.

`os.`**`EX_DATAERR`**

Code de sortie signifiant que les données en entrées étaient incorrectes.

Availability : Unix, not WASI.

os.**EX_NOINPUT**

Code de sortie signifiant qu'un des fichiers d'entrée n'existe pas ou n'est pas lisible.

Availability : Unix, not WASI.

os.**EX_NOUSER**

Code de sortie signifiant qu'un utilisateur spécifié n'existe pas.

Availability : Unix, not WASI.

os.**EX_NOHOST**

Code de sortie signifiant qu'un hôte spécifié n'existe pas.

Availability : Unix, not WASI.

os.**EX_UNAVAILABLE**

Code de sortie signifiant qu'un service requis n'est pas disponible.

Availability : Unix, not WASI.

os.**EX_SOFTWARE**

Code de sortie signifiant qu'une erreur interne d'un programme a été détectée.

Availability : Unix, not WASI.

os.**EX_OSERR**

Code de sortie signifiant qu'une erreur du système d'exploitation a été détectée, comme l'incapacité à réaliser un *fork* ou à créer un tuyau (*pipe*).

Availability : Unix, not WASI.

os.**EX_OSFILE**

Code de sortie signifiant qu'un fichier n'existe pas, n'a pas pu être ouvert, ou avait une autre erreur.

Availability : Unix, not WASI.

os.**EX_CANTCREAT**

Code de sortie signifiant qu'un fichier spécifié par l'utilisateur n'a pas pu être créé.

Availability : Unix, not WASI.

os.**EX_IOERR**

Code de sortie signifiant qu'une erreur est apparue pendant une E/S sur un fichier.

Availability : Unix, not WASI.

os.**EX_TEMPFAIL**

Code de sortie signifiant qu'un échec temporaire est apparu. Cela indique quelque chose qui pourrait ne pas être une erreur, comme une connexion au réseau qui n'a pas pu être établie pendant une opération réessayable.

Availability : Unix, not WASI.

os.**EX_PROTOCOL**

Code de sortie signifiant qu'un protocole d'échange est illégal, invalide, ou non-compris.

Availability : Unix, not WASI.

os.**EX_NOPERM**

Code de sortie signifiant qu'il manque certaines permissions pour réaliser une opération (mais n'est pas destiné au problèmes de système de fichiers).

Availability : Unix, not WASI.

os.**EX_CONFIG**

Code de sortie signifiant qu'une erreur de configuration est apparue.

Availability : Unix, not WASI.

os.**EX_NOTFOUND**

Code de sortie signifiant quelque chose comme « une entrée n'a pas été trouvée ».

Availability : Unix, not WASI.

os.**fork**()

Fork un processus fils. Renvoie `0` dans le processus fils et le PID du processus fils dans le processus père. Si une erreur apparaît, une `OSError` est levée.

Note that some platforms including FreeBSD <= 6.3 and Cygwin have known issues when using `fork()` from a thread.

Lève un *évènement d'audit* `sys.fork` sans argument.

> **Avertissement**
>
> If you use TLS sockets in an application calling `fork()`, see the warning in the `ssl` documentation.

> **Avertissement**
>
> On macOS the use of this function is unsafe when mixed with using higher-level system APIs, and that includes using `urllib.request`.

Modifié dans la version 3.8 : Calling `fork()` in a subinterpreter is no longer supported (`RuntimeError` is raised).

Modifié dans la version 3.12 : If Python is able to detect that your process has multiple threads, `os.fork()` now raises a `DeprecationWarning`.

We chose to surface this as a warning, when detectable, to better inform developers of a design problem that the POSIX platform specifically notes as not supported. Even in code that *appears* to work, it has never been safe to mix threading with `os.fork()` on POSIX platforms. The CPython runtime itself has always made API calls that are not safe for use in the child process when threads existed in the parent (such as `malloc` and `free`).

Users of macOS or users of libc or malloc implementations other than those typically found in glibc to date are among those already more likely to experience deadlocks running such code.

See this discussion on fork being incompatible with threads for technical details of why we're surfacing this longstanding platform compatibility problem to developers.

Availability : POSIX, not WASI, not Android, not iOS.

os.**forkpty**()

Fork un processus fils, en utilisant un nouveau pseudo-terminal comme terminal contrôlant le fils. Renvoie une paire `(pid, fd)` où *pid* vaut `0` dans le fils et le PID du processus fils dans le parent, et *fd* est le descripteur de fichier de la partie maître du pseudo-terminal. Pour une approche plus portable, utilisez le module `pty`. Si une erreur apparaît, une `OSError` est levée.

Lève un *évènement d'audit* `sys.forkpty` sans argument.

> **Avertissement**
>
> On macOS the use of this function is unsafe when mixed with using higher-level system APIs, and that includes using `urllib.request`.

Modifié dans la version 3.8 : Calling `forkpty()` in a subinterpreter is no longer supported (`RuntimeError` is raised).

Modifié dans la version 3.12 : If Python is able to detect that your process has multiple threads, this now raises a `DeprecationWarning`. See the longer explanation on `os.fork()`.

Availability : Unix, not WASI, not Android, not iOS.

`os.kill`(*pid*, *sig*, /)

Envoie le signal *sig* au processus *pid*. Les constantes pour les signaux spécifiques à la plate-forme hôte sont définies dans le module `signal`.

Windows : The `signal.CTRL_C_EVENT` and `signal.CTRL_BREAK_EVENT` signals are special signals which can only be sent to console processes which share a common console window, e.g., some subprocesses. Any other value for *sig* will cause the process to be unconditionally killed by the TerminateProcess API, and the exit code will be set to *sig*.

Voir également `signal.pthread_kill()`.

Lève un *événement d'audit* `os.kill` avec les arguments `pid`, `sig`.

Availability : Unix, Windows, not WASI, not iOS.

Modifié dans la version 3.2 : Prise en charge de Windows.

`os.killpg`(*pgid*, *sig*, /)

Envoie le signal *sig* au groupe de processus *pgid*.

Lève un *événement d'audit* `os.killpg` avec les arguments `pgid`, `sig`.

Availability : Unix, not WASI, not iOS.

`os.nice`(*increment*, /)

Ajoute *increment* à la priorité du processus. Renvoie la nouvelle priorité.

Availability : Unix, not WASI.

`os.pidfd_open`(*pid*, *flags=0*)

Return a file descriptor referring to the process *pid* with *flags* set. This descriptor can be used to perform process management without races and signals.

See the `pidfd_open(2)` man page for more details.

Availability : Linux >= 5.3, Android >= `build-time` API level 31

Ajouté dans la version 3.9.

`os.PIDFD_NONBLOCK`

This flag indicates that the file descriptor will be non-blocking. If the process referred to by the file descriptor has not yet terminated, then an attempt to wait on the file descriptor using `waitid(2)` will immediately return the error `EAGAIN` rather than blocking.

Availability : Linux >= 5.10

Ajouté dans la version 3.12.

`os.plock`(*op*, /)

Verrouille les segments du programme en mémoire. La valeur de *op* (définie dans `<sys/lock.h>`) détermine quels segments sont verrouillés.

Availability : Unix, not WASI, not iOS.

`os.popen`(*cmd*, *mode='r'*, *buffering=-1*)

Open a pipe to or from command *cmd*. The return value is an open file object connected to the pipe, which can be read or written depending on whether *mode* is `'r'` (default) or `'w'`. The *buffering* argument have the same meaning as the corresponding argument to the built-in `open()` function. The returned file object reads or writes text strings rather than bytes.

La méthode `close` renvoie `None` si le sous-processus s'est terminé avec succès, ou le code de retour du sous-processus s'il y a eu une erreur. Sur les systèmes POSIX, si le code de retour est positif, il représente la valeur de retour du processus décalée d'un byte sur la gauche. Si le code de retour est négatif, le processus le processus s'est terminé avec le signal donné par la négation de la valeur de retour. (Par exemple, la valeur de retour pourrait être `-signal.SIGKILL` si le sous-processus a été tué). Sur les systèmes Windows, la valeur de retour contient le code de retour du processus fils dans un entier signé .

On Unix, `waitstatus_to_exitcode()` can be used to convert the `close` method result (exit status) into an exit code if it is not `None`. On Windows, the `close` method result is directly the exit code (or `None`).

Ceci est implémenté en utilisant `subprocess.Popen`. Lisez la documentation de cette classe pour des méthodes plus puissantes pour gérer et communiquer avec des sous-processus.

Availability : not WASI, not Android, not iOS.

> **Note**
>
> The *Python UTF-8 Mode* affects encodings used for *cmd* and pipe contents.
>
> `popen()` is a simple wrapper around `subprocess.Popen`. Use `subprocess.Popen` or `subprocess.run()` to control options like encodings.

Obsolète depuis la version 3.14 : The function is *soft deprecated* and should no longer be used to write new code. The `subprocess` module is recommended instead.

`os.`**`posix_spawn`**(*path*, *argv*, *env*, ***, *file_actions=None*, *setpgroup=None*, *resetids=False*, *setsid=False*, *setsigmask=()*, *setsigdef=()*, *scheduler=None*)

Wraps the `posix_spawn()` C library API for use from Python.

Most users should use `subprocess.run()` instead of `posix_spawn()`.

The positional-only arguments *path*, *args*, and *env* are similar to `execve()`. *env* is allowed to be `None`, in which case current process' environment is used.

The *path* parameter is the path to the executable file. The *path* should contain a directory. Use `posix_spawnp()` to pass an executable file without directory.

The *file_actions* argument may be a sequence of tuples describing actions to take on specific file descriptors in the child process between the C library implementation's `fork()` and `exec()` steps. The first item in each tuple must be one of the three type indicator listed below describing the remaining tuple elements :

`os.`**`POSIX_SPAWN_OPEN`**

(`os.POSIX_SPAWN_OPEN`, *fd*, *path*, *flags*, *mode*)

Performs `os.dup2(os.open(path, flags, mode), fd)`.

`os.`**`POSIX_SPAWN_CLOSE`**

(`os.POSIX_SPAWN_CLOSE`, *fd*)

Performs `os.close(fd)`.

`os.`**`POSIX_SPAWN_DUP2`**

(`os.POSIX_SPAWN_DUP2`, *fd*, *new_fd*)

Performs `os.dup2(fd, new_fd)`.

`os.`**`POSIX_SPAWN_CLOSEFROM`**

(`os.POSIX_SPAWN_CLOSEFROM`, *fd*)

Performs `os.closerange(fd, INF)`.

These tuples correspond to the C library `posix_spawn_file_actions_addopen()`, `posix_spawn_file_actions_addclose()`, `posix_spawn_file_actions_adddup2()`, and `posix_spawn_file_actions_addclosefrom_np()` API calls used to prepare for the `posix_spawn()` call itself.

The *setpgroup* argument will set the process group of the child to the value specified. If the value specified is 0, the child's process group ID will be made the same as its process ID. If the value of *setpgroup* is not set, the child will inherit the parent's process group ID. This argument corresponds to the C library `POSIX_SPAWN_SETPGROUP` flag.

If the *resetids* argument is `True` it will reset the effective UID and GID of the child to the real UID and GID of the parent process. If the argument is `False`, then the child retains the effective UID and GID of the parent. In either case, if the set-user-ID and set-group-ID permission bits are enabled on the executable file, their effect will override the setting of the effective UID and GID. This argument corresponds to the C library `POSIX_SPAWN_RESETIDS` flag.

If the *setsid* argument is `True`, it will create a new session ID for `posix_spawn`. *setsid* requires `POSIX_SPAWN_SETSID` or `POSIX_SPAWN_SETSID_NP` flag. Otherwise, `NotImplementedError` is raised.

The *setsigmask* argument will set the signal mask to the signal set specified. If the parameter is not used, then the child inherits the parent's signal mask. This argument corresponds to the C library `POSIX_SPAWN_SETSIGMASK` flag.

The *sigdef* argument will reset the disposition of all signals in the set specified. This argument corresponds to the C library `POSIX_SPAWN_SETSIGDEF` flag.

The *scheduler* argument must be a tuple containing the (optional) scheduler policy and an instance of `sched_param` with the scheduler parameters. A value of `None` in the place of the scheduler policy indicates that is not being provided. This argument is a combination of the C library `POSIX_SPAWN_SETSCHEDPARAM` and `POSIX_SPAWN_SETSCHEDULER` flags.

Lève un *événement d'audit* `os.posix_spawn` avec les arguments `path`, `argv`, `env`.

Ajouté dans la version 3.8.

Modifié dans la version 3.13 : *env* parameter accepts `None`. `os.POSIX_SPAWN_CLOSEFROM` is available on platforms where `posix_spawn_file_actions_addclosefrom_np()` exists.

Availability : Unix, not WASI, not Android, not iOS.

`os.`**`posix_spawnp`**(*path*, *argv*, *env*, ***, *file_actions=None*, *setpgroup=None*, *resetids=False*, *setsid=False*, *setsigmask=()*, *setsigdef=()*, *scheduler=None*)

Wraps the `posix_spawnp()` C library API for use from Python.

Similar to `posix_spawn()` except that the system searches for the *executable* file in the list of directories specified by the `PATH` environment variable (in the same way as for `execvp(3)`).

Lève un *événement d'audit* `os.posix_spawn` avec les arguments `path`, `argv`, `env`.

Ajouté dans la version 3.8.

Availability : POSIX, not WASI, not Android, not iOS.

See `posix_spawn()` documentation.

`os.`**`register_at_fork`**(***, *before=None*, *after_in_parent=None*, *after_in_child=None*)

Enregistre des appelables (*callables*) à exécuter quand un nouveau processus enfant est crée avec `os.fork()` ou des APIs similaires de clonage de processus. Les paramètres sont optionnels et par mots-clé uniquement. Chacun spécifie un point d'appel différent.
— *before* est une fonction appelée avant de *forker* un processus enfant.
— *after_in_parent* est une fonction appelée depuis le processus parent après avoir *forké* un processus enfant.
— *after_in_child* est une fonction appelée depuis le processus enfant.
Ces appels ne sont effectués que si le contrôle est censé retourner à l'interpréteur Python. Un lancement de `subprocess` typique ne les déclenchera pas, car l'enfant ne ré-entre pas dans l'interpréteur.

Les fonctions enregistrées pour l'exécution avant le *fork* sont appelées dans l'ordre inverse à leur enregistrement. Les fonctions enregistrées pour l'exécution après le *fork* (soit dans le parent ou dans l'enfant) sont appelées dans l'ordre de leur enregistrement.

Notez que les appels à `fork()` faits par du code C de modules tiers peuvent ne pas appeler ces fonctions, à moins que ce code appelle explicitement `PyOS_BeforeFork()`, `PyOS_AfterFork_Parent()` et `PyOS_AfterFork_Child()`.

Il n'y a aucun moyen d'annuler l'enregistrement d'une fonction.

Availability : Unix, not WASI, not Android, not iOS.

Ajouté dans la version 3.7.

os.**spawnl**(*mode*, *path*, ...)

os.**spawnle**(*mode*, *path*, ..., *env*)

os.**spawnlp**(*mode*, *file*, ...)

os.**spawnlpe**(*mode*, *file*, ..., *env*)

os.**spawnv**(*mode*, *path*, *args*)

os.**spawnve**(*mode*, *path*, *args*, *env*)

os.**spawnvp**(*mode*, *file*, *args*)

os.**spawnvpe**(*mode*, *file*, *args*, *env*)

Exécute le programme *path* dans un nouveau processus.

(Notez que le module `subprocess` fournit des outils plus puissants pour générer de nouveaux processus et récupérer leur valeur de retour. Il est préférable d'utiliser ce module que ces fonctions. Voyez surtout la section *Remplacer les fonctions plus anciennes par le module subprocess*.)

Si *mode* vaut `P_NOWAIT`, cette fonction renvoie le PID du nouveau processus, et si *mode* vaut `P_WAIT`, la fonction renvoie le code de sortie du processus s'il se termine normalement, ou `-signal` où *signal* est le signal qui a tué le processus. Sur Windows, le PID du processus sera en fait l'identificateur du processus (*process handle*) et peut donc être utilisé avec la fonction `waitpid()`.

Note on VxWorks, this function doesn't return `-signal` when the new process is killed. Instead it raises OSError exception.

The "l" and "v" variants of the `spawn*` functions differ in how command-line arguments are passed. The "l" variants are perhaps the easiest to work with if the number of parameters is fixed when the code is written ; the individual parameters simply become additional parameters to the `spawnl*()` functions. The "v" variants are good when the number of parameters is variable, with the arguments being passed in a list or tuple as the *args* parameter. In either case, the arguments to the child process must start with the name of the command being run.

Les variantes qui incluent un « p » vers la fin (`spawnlp()`, `spawnlpe()`, `spawnvp()`, et `spawnvpe()`) utiliseront la variable d'environnement `PATH` pour localiser le programme *file*. Quand l'environnement est remplacé (en utilisant une des variantes `spawn*e`, discutées dans le paragraphe suivant), le nouvel environnement est utilisé comme source de la variable d'environnement `PATH`. Les autres variantes `spawnl()`, `spawnle()`, `spawnv()`, et `spawnve()` n'utiliseront pas la variable d'environnement `PATH` pour localiser l'exécutable. *path* doit contenir un chemin absolue ou relatif approprié.

Pour les fonctions `spawnle()`, `spawnlpe()`, `spawnve()`, et `spawnvpe()` (notez qu'elles finissent toutes par « e »), le paramètre *env* doit être un *mapping* qui est utilisé pour définir les variables d'environnement du nouveau processus (celles-ci sont utilisées à la place de l'environnement du nouveau processus). Les fonctions `spawnl()`, `spawnlp()`, `spawnv()`, et `spawnvp()` causent toutes un héritage de l'environnement du processus actuel par le processus fils. Notez que les clefs et les valeurs du dictionnaire *env* doivent être des chaînes de caractères. Des valeurs invalides pour les clefs ou les valeurs met la fonction en échec et renvoie `127`.

Par exemple, les appels suivants à `spawnlp()` et `spawnvpe()` sont équivalents :

```
import os
os.spawnlp(os.P_WAIT, 'cp', 'cp', 'index.html', '/dev/null')

L = ['cp', 'index.html', '/dev/null']
os.spawnvpe(os.P_WAIT, 'cp', L, os.environ)
```

Lève un *événement d'audit* `os.spawn` avec les arguments `mode`, `path`, `args`, `env`.

Availability : Unix, Windows, not WASI, not Android, not iOS.

`spawnlp()`, `spawnlpe()`, `spawnvp()`, et `spawnvpe()` ne sont pas disponibles sur Windows. `spawnle()` et `spawnve()` ne sont pas sécurisés pour les appels concurrents (*thread-safe*) sur Windows, il est conseillé

d'utiliser le module `subprocess` à la place.

Modifié dans la version 3.6 : Accepte un *path-like object*.

Obsolète depuis la version 3.14 : These functions are *soft deprecated* and should no longer be used to write new code. The `subprocess` module is recommended instead.

os.**P_NOWAIT**

os.**P_NOWAITO**

Possible values for the *mode* parameter to the `spawn*` family of functions. If either of these values is given, the `spawn*` functions will return as soon as the new process has been created, with the process id as the return value.

Availability : Unix, Windows.

os.**P_WAIT**

Possible value for the *mode* parameter to the `spawn*` family of functions. If this is given as *mode*, the `spawn*` functions will not return until the new process has run to completion and will return the exit code of the process the run is successful, or `-signal` if a signal kills the process.

Availability : Unix, Windows.

os.**P_DETACH**

os.**P_OVERLAY**

Valeurs possibles pour le paramètre *mode* de la famille de fonctions `spawn*`. Ces valeurs sont moins portables que celles listées plus haut. `P_DETACH` est similaire à `P_NOWAIT`, mais le nouveau processus est détaché de la console du processus appelant. Si `P_OVERLAY` est utilisé, le processus actuel sera remplacé. La fonction `spawn*` ne sort jamais.

Availability : Windows.

os.**startfile**(*path*[, *operation*][, *arguments*][, *cwd*][, *show_cmd*])

Lance un fichier avec son application associée.

When *operation* is not specified, this acts like double-clicking the file in Windows Explorer, or giving the file name as an argument to the **start** command from the interactive command shell : the file is opened with whatever application (if any) its extension is associated.

When another *operation* is given, it must be a "command verb" that specifies what should be done with the file. Common verbs documented by Microsoft are `'open'`, `'print'` and `'edit'` (to be used on files) as well as `'explore'` and `'find'` (to be used on directories).

When launching an application, specify *arguments* to be passed as a single string. This argument may have no effect when using this function to launch a document.

The default working directory is inherited, but may be overridden by the *cwd* argument. This should be an absolute path. A relative *path* will be resolved against this argument.

Use *show_cmd* to override the default window style. Whether this has any effect will depend on the application being launched. Values are integers as supported by the Win32 `ShellExecute()` function.

`startfile()` returns as soon as the associated application is launched. There is no option to wait for the application to close, and no way to retrieve the application's exit status. The *path* parameter is relative to the current directory or *cwd*. If you want to use an absolute path, make sure the first character is not a slash (`'/'`) Use `pathlib` or the `os.path.normpath()` function to ensure that paths are properly encoded for Win32.

To reduce interpreter startup overhead, the Win32 `ShellExecute()` function is not resolved until this function is first called. If the function cannot be resolved, `NotImplementedError` will be raised.

Lève un *événement d'audit* `os.startfile` avec les arguments `path`, `operation`.

Raises an *auditing event* `os.startfile/2` with arguments `path`, `operation`, `arguments`, `cwd`, `show_cmd`.

Availability : Windows.

Modifié dans la version 3.10 : Added the *arguments*, *cwd* and *show_cmd* arguments, and the `os.startfile/2` audit event.

`os.system(command)`

Execute the command (a string) in a subshell. This is implemented by calling the Standard C function `system()`, and has the same limitations. Changes to `sys.stdin`, etc. are not reflected in the environment of the executed command. If *command* generates any output, it will be sent to the interpreter standard output stream. The C standard does not specify the meaning of the return value of the C function, so the return value of the Python function is system-dependent.

On Unix, the return value is the exit status of the process encoded in the format specified for `wait()`.

Sur Windows, la valeur de retour est celle renvoyée par l'invite de commande système après avoir lancé *command*. L'invite de commande est donné par la variable d'environnement Windows `COMSPEC`. Elle vaut habituellement **`cmd.exe`**, qui renvoie l'état de sortie de la commande lancée. Sur les systèmes qui utilisent un invite de commande non-natif, consultez la documentation propre à l'invite.

The `subprocess` module provides more powerful facilities for spawning new processes and retrieving their results ; using that module is recommended to using this function. See the *Remplacer les fonctions plus anciennes par le module subprocess* section in the `subprocess` documentation for some helpful recipes.

On Unix, `waitstatus_to_exitcode()` can be used to convert the result (exit status) into an exit code. On Windows, the result is directly the exit code.

Lève un *évènement d'audit* `os.system` avec comme argument `command`.

Availability : Unix, Windows, not WASI, not Android, not iOS.

`os.times()`

Renvoie les temps globaux actuels d'exécution du processus. La valeur de retour est un objet avec cinq attributs :
— `user` - user time
— `system` - system time
— `children_user` - user time of all child processes
— `children_system` - system time of all child processes
— `elapsed` - elapsed real time since a fixed point in the past
For backwards compatibility, this object also behaves like a five-tuple containing `user`, `system`, `children_user`, `children_system`, and `elapsed` in that order.

See the Unix manual page `times(2)` and times(3) manual page on Unix or the GetProcessTimes MSDN on Windows. On Windows, only `user` and `system` are known ; the other attributes are zero.

Availability : Unix, Windows.

Modifié dans la version 3.3 : Type de retour changé d'un quintuplet en un objet compatible avec le type *n*-uplet, avec des attributs nommés.

`os.wait()`

Attend qu'un processus fils soit complété, et renvoie une paire contenant son PID et son état de sortie : un nombre de 16 bits dont le *byte* de poids faible est le nombre correspondant au signal qui a tué le processus, et dont le *byte* de poids fort est le statut de sortie (si la signal vaut `0`). Le bit de poids fort du *byte* de poids faible est mis à 1 si un (fichier système) *core file* a été produit.

If there are no children that could be waited for, `ChildProcessError` is raised.

`waitstatus_to_exitcode()` can be used to convert the exit status into an exit code.

Availability : Unix, not WASI, not Android, not iOS.

> **Voir aussi**
>
> The other `wait*()` functions documented below can be used to wait for the completion of a specific child process and have more options. `waitpid()` is the only one also available on Windows.

`os.waitid(idtype, id, options, /)`

Wait for the completion of a child process.

idtype can be `P_PID`, `P_PGID`, `P_ALL`, or (on Linux) `P_PIDFD`. The interpretation of *id* depends on it ; see their individual descriptions.

options is an OR combination of flags. At least one of `WEXITED`, `WSTOPPED` or `WCONTINUED` is required ; `WNOHANG` and `WNOWAIT` are additional optional flags.

The return value is an object representing the data contained in the `siginfo_t` structure with the following attributes :
— `si_pid` (process ID)
— `si_uid` (real user ID of the child)
— `si_signo` (always `SIGCHLD`)
— `si_status` (the exit status or signal number, depending on `si_code`)
— `si_code` (see `CLD_EXITED` for possible values)
If `WNOHANG` is specified and there are no matching children in the requested state, `None` is returned. Otherwise, if there are no matching children that could be waited for, `ChildProcessError` is raised.

Availability : Unix, not WASI, not Android, not iOS.

Ajouté dans la version 3.3.

Modifié dans la version 3.13 : This function is now available on macOS as well.

`os.waitpid(pid, options, /)`

Les détails de cette fonction diffèrent sur Unix et Windows.

Sur Unix : attend qu'un processus fils donné par son id de processus *pid* soit terminé, et renvoie une paire contenant son PID et son statut de sortie (encodé comme pour `wait()`). La sémantique de cet appel est affecté par la valeur de l'argument entier *options*, qui devrait valoir `0` pour les opérations normales.

Si *pid* est plus grand que `0`, `waitpid()` introduit une requête pour des informations sur le statut du processus spécifié. Si *pid* vaut `0`, la requête est pour le statut de tous les fils dans le groupe de processus du processus actuel. Si *pid* vaut `-1`, la requête concerne tous les processus fils du processus actuel. Si *pid* est inférieur à `-1`, une requête est faite pour le statut de chaque processus du groupe de processus donné par `-pid` (la valeur absolue de *pid*).

options is an OR combination of flags. If it contains `WNOHANG` and there are no matching children in the requested state, `(0, 0)` is returned. Otherwise, if there are no matching children that could be waited for, `ChildProcessError` is raised. Other options that can be used are `WUNTRACED` and `WCONTINUED`.

Sur Windows : attend que qu'un processus donné par l'identificateur de processus (*process handle*) *pid* soit complété, et renvoie une paire contenant *pid* et son statut de sortie décalé de 8 bits vers la gauche (le décalage rend l'utilisation multiplateformes plus facile). Un *pid* inférieur ou égal à `0` n'a aucune signification particulière en Windows, et lève une exception. L'argument entier *options* n'a aucun effet. *pid* peut faire référence à tout processus dont l'identifiant est connu et pas nécessairement à un processus fils. Les fonctions `spawn*` appelées avec `P_NOWAIT` renvoient des identificateurs de processus appropriés.

`waitstatus_to_exitcode()` can be used to convert the exit status into an exit code.

Availability : Unix, Windows, not WASI, not Android, not iOS.

Modifié dans la version 3.5 : Si l'appel système est interrompu et que le gestionnaire de signal ne lève aucune exception, la fonction réessaye l'appel système au lieu de lever une `InterruptedError` (voir la **PEP 475** à propos du raisonnement).

`os.wait3(options)`

Similar to `waitpid()`, except no process id argument is given and a 3-element tuple containing the child's process id, exit status indication, and resource usage information is returned. Refer to `resource.getrusage()` for details on resource usage information. The *options* argument is the same as that provided to `waitpid()` and `wait4()`.

`waitstatus_to_exitcode()` can be used to convert the exit status into an exitcode.

Availability : Unix, not WASI, not Android, not iOS.

os.**wait4**(*pid*, *options*)

Similar to `waitpid()`, except a 3-element tuple, containing the child's process id, exit status indication, and resource usage information is returned. Refer to `resource.getrusage()` for details on resource usage information. The arguments to `wait4()` are the same as those provided to `waitpid()`.

`waitstatus_to_exitcode()` can be used to convert the exit status into an exitcode.

Availability : Unix, not WASI, not Android, not iOS.

os.**P_PID**

os.**P_PGID**

os.**P_ALL**

os.**P_PIDFD**

These are the possible values for *idtype* in `waitid()`. They affect how *id* is interpreted :

— `P_PID` - wait for the child whose PID is *id*.
— `P_PGID` - wait for any child whose progress group ID is *id*.
— `P_ALL` - wait for any child ; *id* is ignored.
— `P_PIDFD` - wait for the child identified by the file descriptor *id* (a process file descriptor created with `pidfd_open()`).

Availability : Unix, not WASI, not Android, not iOS.

> **Note**
>
> `P_PIDFD` is only available on Linux >= 5.4.

Ajouté dans la version 3.3.

Ajouté dans la version 3.9 : The `P_PIDFD` constant.

os.**WCONTINUED**

This *options* flag for `waitpid()`, `wait3()`, `wait4()`, and `waitid()` causes child processes to be reported if they have been continued from a job control stop since they were last reported.

Availability : Unix, not WASI, not Android, not iOS.

os.**WEXITED**

This *options* flag for `waitid()` causes child processes that have terminated to be reported.

The other `wait*` functions always report children that have terminated, so this option is not available for them.

Availability : Unix, not WASI, not Android, not iOS.

Ajouté dans la version 3.3.

os.**WSTOPPED**

This *options* flag for `waitid()` causes child processes that have been stopped by the delivery of a signal to be reported.

This option is not available for the other `wait*` functions.

Availability : Unix, not WASI, not Android, not iOS.

Ajouté dans la version 3.3.

os.**WUNTRACED**

This *options* flag for `waitpid()`, `wait3()`, and `wait4()` causes child processes to also be reported if they have been stopped but their current state has not been reported since they were stopped.

This option is not available for `waitid()`.

Availability : Unix, not WASI, not Android, not iOS.

os.**WNOHANG**

This *options* flag causes `waitpid()`, `wait3()`, `wait4()`, and `waitid()` to return right away if no child process status is available immediately.

Availability : Unix, not WASI, not Android, not iOS.

os.**WNOWAIT**

This *options* flag causes `waitid()` to leave the child in a waitable state, so that a later `wait*()` call can be used to retrieve the child status information again.

This option is not available for the other `wait*` functions.

Availability : Unix, not WASI, not Android, not iOS.

os.**CLD_EXITED**

os.**CLD_KILLED**

os.**CLD_DUMPED**

os.**CLD_TRAPPED**

os.**CLD_STOPPED**

os.**CLD_CONTINUED**

These are the possible values for `si_code` in the result returned by `waitid()`.

Availability : Unix, not WASI, not Android, not iOS.

Ajouté dans la version 3.3.

Modifié dans la version 3.9 : Added `CLD_KILLED` and `CLD_STOPPED` values.

os.**waitstatus_to_exitcode**(*status*)

Convert a wait status to an exit code.

On Unix :

— If the process exited normally (if `WIFEXITED(status)` is true), return the process exit status (return `WEXITSTATUS(status)`) : result greater than or equal to 0.

— If the process was terminated by a signal (if `WIFSIGNALED(status)` is true), return `-signum` where *signum* is the number of the signal that caused the process to terminate (return `-WTERMSIG(status)`) : result less than 0.

— Otherwise, raise a `ValueError`.

On Windows, return *status* shifted right by 8 bits.

On Unix, if the process is being traced or if `waitpid()` was called with `WUNTRACED` option, the caller must first check if `WIFSTOPPED(status)` is true. This function must not be called if `WIFSTOPPED(status)` is true.

> **Voir aussi**
>
> `WIFEXITED()`, `WEXITSTATUS()`, `WIFSIGNALED()`, `WTERMSIG()`, `WIFSTOPPED()`, `WSTOPSIG()` functions.

Availability : Unix, Windows, not WASI, not Android, not iOS.

Ajouté dans la version 3.9.

Les fonctions suivantes prennent un code de statu tel que renvoyé par `system()`, `wait()`, ou `waitpid()` en paramètre. Ils peuvent être utilisés pour déterminer la disposition d'un processus.

os.**WCOREDUMP**(*status*, /)

Renvoie `True` si un vidage système (*core dump*) a été généré pour le processus, sinon, renvoie `False`.

This function should be employed only if `WIFSIGNALED()` is true.

Availability : Unix, not WASI, not Android, not iOS.

os.**WIFCONTINUED**(*status*)

Return `True` if a stopped child has been resumed by delivery of `SIGCONT` (if the process has been continued from a job control stop), otherwise return `False`.

See `WCONTINUED` option.

Availability : Unix, not WASI, not Android, not iOS.

os.**WIFSTOPPED**(*status*)

Return `True` if the process was stopped by delivery of a signal, otherwise return `False`.

`WIFSTOPPED()` only returns `True` if the `waitpid()` call was done using `WUNTRACED` option or when the process is being traced (see `ptrace(2)`).

Availability : Unix, not WASI, not Android, not iOS.

os.**WIFSIGNALED**(*status*)

Return `True` if the process was terminated by a signal, otherwise return `False`.

Availability : Unix, not WASI, not Android, not iOS.

os.**WIFEXITED**(*status*)

Return `True` if the process exited terminated normally, that is, by calling `exit()` or `_exit()`, or by returning from `main()` ; otherwise return `False`.

Availability : Unix, not WASI, not Android, not iOS.

os.**WEXITSTATUS**(*status*)

Return the process exit status.

This function should be employed only if `WIFEXITED()` is true.

Availability : Unix, not WASI, not Android, not iOS.

os.**WSTOPSIG**(*status*)

Renvoie le signal qui a causé l'arrêt du processus.

This function should be employed only if `WIFSTOPPED()` is true.

Availability : Unix, not WASI, not Android, not iOS.

os.**WTERMSIG**(*status*)

Return the number of the signal that caused the process to terminate.

This function should be employed only if `WIFSIGNALED()` is true.

Availability : Unix, not WASI, not Android, not iOS.

16.1.8 Interface pour l'ordonnanceur

Ces fonctions contrôlent un processus se voit allouer du temps de processus par le système d'exploitation. Elles ne sont disponibles que sur certaines plate-formes Unix. Pour des informations plus détaillées, consultez les pages de manuels Unix.

Ajouté dans la version 3.3.

Les polices d'ordonnancement suivantes sont exposées si elles sont gérées par le système d'exploitation.

os.**SCHED_OTHER**

La police d'ordonnancement par défaut.

os.**SCHED_BATCH**

Police d'ordonnancement pour les processus intensifs en utilisation du processeur. Cette police essaye de préserver l'interactivité pour le reste de l'ordinateur.

os.**SCHED_DEADLINE**

Scheduling policy for tasks with deadline constraints.

Ajouté dans la version 3.14.

os.**SCHED_IDLE**

Police d'ordonnancement pour les tâches de fond avec une priorité extrêmement faible.

os.**SCHED_NORMAL**

Alias for `SCHED_OTHER`.

Ajouté dans la version 3.14.

os.**SCHED_SPORADIC**

Police d'ordonnancement pour des programmes serveurs sporadiques.

os.**SCHED_FIFO**

Une police d'ordonnancement *FIFO* (dernier arrivé, premier servi).

os.**SCHED_RR**

Une police d'ordonnancement *round-robin* (tourniquet).

os.**SCHED_RESET_ON_FORK**

Cette option peut combiner différentes politiques d'ordonnancement avec un OU bit-à-bit. Quand un processus avec cette option se dédouble, la politique d'ordonnancement et la priorité du processus fils sont remises aux valeurs par défaut.

class os.**sched_param**(*sched_priority*)

Cette classe représente des paramètres d'ordonnancement réglables utilisés pour `sched_setparam()`, `sched_setscheduler()`, et `sched_getparam()`. Un objet de ce type est immuable.

Pour le moment, il n'y a qu'un seul paramètre possible :

sched_priority

La priorité d'ordonnancement pour une police d'ordonnancement.

os.**sched_get_priority_min**(*policy*)

Récupère la valeur minimum pour une priorité pour la police *policy*. *policy* est une des constantes de police définies ci-dessus.

os.**sched_get_priority_max**(*policy*)

Récupère la valeur maximum pour une priorité pour la police *policy*. *policy* est une des constantes de police définies ci-dessus.

os.**sched_setscheduler**(*pid*, *policy*, *param*, /)

Définit la police d'ordonnancement pour le processus de PID *pid*. Un *pid* de `0` signifie le processus appelant. *policy* est une des constantes de police définies ci-dessus. *param* est une instance de la classe `sched_param`.

os.**sched_getscheduler**(*pid*, /)

Renvoie la police d'ordonnancement pour le processus de PID *pid*. Un *pid* de 0 signifie le processus appelant. Le résultat est une des constantes de police définies ci-dessus.

os.**sched_setparam**(*pid*, *param*, /)

Set the scheduling parameters for the process with PID *pid*. A *pid* of 0 means the calling process. *param* is a `sched_param` instance.

os.**sched_getparam**(*pid*, /)

Renvoie les paramètres d'ordonnancement dans une de `sched_param` pour le processus de PID *pid*. Un *pid* de `0` signifie le processus appelant.

os.**sched_rr_get_interval**(*pid*, /)

Renvoie le quantum de temps du *round-robin* (en secondes) pour le processus de PID *pid*. Un *pid* de `0` signifie le processus appelant.

os.**sched_yield**()

Voluntarily relinquish the CPU. See *sched_yield(2)* for details.

os.**sched_setaffinity**(*pid*, *mask*, /)

Restreint le processus de PID *pid* (ou le processus actuel si *pid* vaut 0) à un ensemble de CPUs. *mask* est un itérable d'entiers représentant l'ensemble de CPUs auquel le processus doit être restreint.

os.**sched_getaffinity**(*pid*, /)

Return the set of CPUs the process with PID *pid* is restricted to.

If *pid* is zero, return the set of CPUs the calling thread of the current process is restricted to.

See also the *process_cpu_count()* function.

16.1.9 Diverses informations sur le système

os.**confstr**(*name*, /)

Renvoie les valeurs de configuration en chaînes de caractères. *name* spécifie la valeur de configuration à récupérer. Ce peut être une chaîne de caractères représentant le nom d'une valeur système définie dans un nombre de standards (POSIX, Unix 95, Unix 98, et d'autres). Certaines plate-formes définissent des noms supplémentaires également. Les noms connus par le système d'exploitation hôte sont données dans les clefs du dictionnaire `confstr_names`. Pour les variables de configuration qui ne sont pas incluses dans ce *mapping*, passer un entier pour *name* est également accepté.

Si la valeur de configuration spécifiée par *name* n'est pas définie, `None` est renvoyé.

Si *name* est une chaîne de caractères et n'est pas connue, une *ValueError* est levée. Si une valeur spécifique pour *name* n'est pas gérée par le système hôte, même si elle est incluse dans `confstr_names`, une *OSError* est levée avec *errno.EINVAL* pour numéro d'erreur.

Availability : Unix.

os.**confstr_names**

Dictionnaire liant les noms acceptés par *confstr()* aux valeurs entières définies pour ces noms par le système d'exploitation hôte. Cela peut être utilisé pour déterminer l'ensemble des noms connus du système.

Availability : Unix.

os.**cpu_count**()

Return the number of logical CPUs in the **system**. Returns `None` if undetermined.

The *process_cpu_count()* function can be used to get the number of logical CPUs usable by the calling thread of the **current process**.

Ajouté dans la version 3.4.

Modifié dans la version 3.13 : If `-X cpu_count` is given or `PYTHON_CPU_COUNT` is set, *cpu_count()* returns the overridden value *n*.

os.**getloadavg**()

Renvoie le nombre de processus dans la file d'exécution du système en moyenne dans les dernières 1, 5, et 15 minutes, ou lève une *OSError* si la charge moyenne est impossible à récupérer.

Availability : Unix.

os.**process_cpu_count**()

Get the number of logical CPUs usable by the calling thread of the **current process**. Returns `None` if undetermined. It can be less than *cpu_count()* depending on the CPU affinity.

The *cpu_count()* function can be used to get the number of logical CPUs in the **system**.

If `-X cpu_count` is given or `PYTHON_CPU_COUNT` is set, *process_cpu_count()* returns the overridden value *n*.

See also the *sched_getaffinity()* function.

Ajouté dans la version 3.13.

os.**sysconf**(*name*, /)

Renvoie les valeurs de configuration en nombres entiers. Si la valeur de configuration définie par *name* n'est pas spécifiée, `-1` est renvoyé. Les commentaires concernant le paramètre *name* de `confstr()` s'appliquent également ici, le dictionnaire qui fournit les informations sur les noms connus est donné par `sysconf_names`.

Availability : Unix.

os.**sysconf_names**

Dictionnaire liant les noms acceptés par `sysconf()` aux valeurs entières définies pour ces noms par le système d'exploitation hôte. Cela peut être utilisé pour déterminer l'ensemble des noms connus du système.

Availability : Unix.

Modifié dans la version 3.11 : Add `'SC_MINSIGSTKSZ'` name.

Les valeurs suivantes sont utilisées pour gérer les opérations de manipulations de chemins. Elles sont définies pour toutes les plate-formes.

Des opérations de plus haut niveau sur les chemins sont définies dans le module `os.path`.

os.**curdir**

La chaîne de caractère constante utilisée par le système d'exploitation pour référencer le répertoire actuel. Ça vaut `'.'` pour Windows et POSIX. Également disponible par `os.path`.

os.**pardir**

La chaîne de caractère constante utilisée par le système d'exploitation pour référencer le répertoire parent. Ça vaut `'..'` pour Windows et POSIX. Également disponible par `os.path`.

os.**sep**

Le caractère utilisé par le système d'exploitation pour séparer les composantes des chemins. C'est `'/'` pour POSIX, et `'\\'` pour Windows. Notez que ce n'est pas suffisant pour pouvoir analyser ou concaténer correctement des chemins (utilisez alors `os.path.split()` et `os.path.join()`), mais ça peut s'avérer utile occasionnellement. Également disponible par `os.path`.

os.**altsep**

Un caractère alternatif utilisé par le système d'exploitation pour séparer les composantes des chemins, ou `None` si un seul séparateur existe. Ça vaut `'/'` sur Windows où `sep` est un antislash `'\'`. Également disponible par `os.path`.

os.**extsep**

Le caractère qui sépare la base du nom d'un fichier et son extension. Par exemple, le `'.'` de `os.py`. Également disponible par `os.path`.

os.**pathsep**

Le caractère conventionnellement utilisé par le système d'exploitation pour séparer la recherche des composantes de chemin (comme dans la variable d'environnement `PATH`). Cela vaut `':'` pour POSIX, ou `';'` pour Windows. Également disponible par `os.path`.

os.**defpath**

Le chemin de recherche par défaut utilisé par `exec*` et `spawn*` si l'environnement n'a pas une clef `'PATH'`. Également disponible par `os.path`.

os.**linesep**

La chaîne de caractères utilisée pour séparer (ou plutôt pour terminer) les lignes sur la plate-forme actuelle. Ce peut être un caractère unique (comme `'\n'` pour POSIX,) ou plusieurs caractères (comme `'\r\n'` pour Windows). N'utilisez pas *os.linesep* comme terminateur de ligne quand vous écrivez dans un fichier ouvert en mode *texte* (par défaut). Utilisez un unique `'\n'` à la place, sur toutes les plate-formes.

os.**devnull**

Le chemin de fichier du périphérique *null*. Par exemple : `'/dev/null'` pour POSIX, `'nul'` pour Windows. Également disponible par `os.path`.

os.**RTLD_LAZY**

os.**RTLD_NOW**

os.**RTLD_GLOBAL**

os.**RTLD_LOCAL**

os.**RTLD_NODELETE**

os.**RTLD_NOLOAD**

os.**RTLD_DEEPBIND**

Marqueurs à utiliser avec les fonctions `setdlopenflags()` et `getdlopenflags()`. Voir les pages de manuel Unix `dlopen(3)` pour les différences de significations entre les marqueurs.

Ajouté dans la version 3.3.

16.1.10 Nombres aléatoires

os.**getrandom**(*size*, *flags=0*)

Obtient *size* octets aléatoires. La fonction renvoie éventuellement moins d'octets que demandé.

Ces octets peuvent être utilisés pour initialiser un générateur de nombres aléatoires dans l'espace utilisateur ou pour des raisons cryptographiques.

`getrandom()` se base sur l'entropie rassemblée depuis les pilotes des périphériques et autres sources de bruits de l'environnement. La lecture de grosses quantités de données aura un impact négatif sur les autres utilisateurs des périphériques `/dev/random` et `/dev/urandom`.

The flags argument is a bit mask that can contain zero or more of the following values ORed together : `os.GRND_RANDOM` and `GRND_NONBLOCK`.

See also the Linux getrandom() manual page.

Availability : Linux >= 3.17.

Ajouté dans la version 3.6.

os.**urandom**(*size*, /)

Return a bytestring of *size* random bytes suitable for cryptographic use.

Cette fonction renvoie des octets aléatoires depuis un source spécifique à l'OS. Les données renvoyées sont censées être suffisamment imprévisibles pour les applications cryptographiques, bien que la qualité dépende de l'implémentation du système.

Sous Linux, si l'appel système `getrandom()` est disponible, il est utilisé en mode bloquant : il bloque jusqu'à ce que la réserve d'entropie d'*urandom* soit initialisée (128 bits d'entropie sont collectés par le noyau). Voir la **PEP 524** pour plus d'explications. Sous Linux, la fonction `getrandom()` peut être utilisée pour obtenir des octets aléatoires en mode non-bloquant (avec l'option `GRND_NONBLOCK`) ou attendre jusqu'à ce que la réserve d'entropie d'*urandom* soit initialisée.

Sur un système de type UNIX, les octets aléatoires sont lus depuis le périphérique `/dev/urandom`. Si le périphérique `/dev/urandom` n'est pas disponible ou n'est pas lisible, l'exception `NotImplementedError` est levée.

On Windows, it will use `BCryptGenRandom()`.

> **Voir aussi**
>
> Le module `secrets` fournit des fonctions de plus haut niveau. Pour une interface facile à utiliser du générateur de nombres aléatoires fourni par votre plate-forme, veuillez regarder `random.SystemRandom`.

Modifié dans la version 3.5 : Sur Linux 3.17 et plus récent, l'appel système `getrandom()` est maintenant utilisé quand il est disponible. Sur OpenBSD 5.6 et plus récent, la fonction C `getentropy()` est utilisée. Ces fonctions évitent l'utilisation interne d'un descripteur de fichier.

Modifié dans la version 3.5.2 : Sous Linux, si l'appel système `getrandom()` bloque (la réserve d'entropie d'*urandom* n'est pas encore initialisée), réalise à la place une lecture de `/dev/urandom`.

Modifié dans la version 3.6 : Sous Linux, `getrandom()` est maintenant utilisé en mode bloquant pour renforcer la sécurité.

Modifié dans la version 3.11 : On Windows, `BCryptGenRandom()` is used instead of `CryptGenRandom()` which is deprecated.

`os.GRND_NONBLOCK`

Par défaut, quand elle lit depuis `/dev/random`, `getrandom()` bloque si aucun octet aléatoire n'est disponible, et quand elle lit depuis `/dev/urandom`, elle bloque si la réserve d'entropie n'a pas encore été initialisée.

Si l'option `GRND_NONBLOCK` est activée, `getrandom()` ne bloque pas dans ces cas, mais lève immédiatement une `BlockingIOError`.

Ajouté dans la version 3.6.

`os.GRND_RANDOM`

Si ce bit est activé, les octets aléatoires sont puisés depuis `/dev/random` plutôt que `/dev/urandom`.

Ajouté dans la version 3.6.

16.2 `io` --- Core tools for working with streams

Code source : Lib/io.py

16.2.1 Aperçu

The `io` module provides Python's main facilities for dealing with various types of I/O. There are three main types of I/O : *text I/O*, *binary I/O* and *raw I/O*. These are generic categories, and various backing stores can be used for each of them. A concrete object belonging to any of these categories is called a *file object*. Other common terms are *stream* and *file-like object*.

Independent of its category, each concrete stream object will also have various capabilities : it can be read-only, write-only, or read-write. It can also allow arbitrary random access (seeking forwards or backwards to any location), or only sequential access (for example in the case of a socket or pipe).

All streams are careful about the type of data you give to them. For example giving a `str` object to the `write()` method of a binary stream will raise a `TypeError`. So will giving a `bytes` object to the `write()` method of a text stream.

Modifié dans la version 3.3 : Operations that used to raise `IOError` now raise `OSError`, since `IOError` is now an alias of `OSError`.

Entrée/sortie de texte

Text I/O expects and produces `str` objects. This means that whenever the backing store is natively made of bytes (such as in the case of a file), encoding and decoding of data is made transparently as well as optional translation of platform-specific newline characters.

The easiest way to create a text stream is with `open()`, optionally specifying an encoding :

```
f = open("myfile.txt", "r", encoding="utf-8")
```

Les flux de texte en mémoire sont également disponible sous forme d'objets `StringIO` :

```
f = io.StringIO("some initial text data")
```

> **Note**
>
> When working with a non-blocking stream, be aware that read operations on text I/O objects might raise a `BlockingIOError` if the stream cannot perform the operation immediately.

L'API de flux textuel est décrite en détail dans la documentation de la classe `TextIOBase`.

Binary I/O

Binary I/O (also called *buffered I/O*) expects *bytes-like objects* and produces `bytes` objects. No encoding, decoding, or newline translation is performed. This category of streams can be used for all kinds of non-text data, and also when manual control over the handling of text data is desired.

The easiest way to create a binary stream is with `open()` with `'b'` in the mode string :

```
f = open("myfile.jpg", "rb")
```

Les flux de texte en mémoire sont également disponible sous forme d'objets `BytesIO` :

```
f = io.BytesIO(b"some initial binary data: \x00\x01")
```

L'API du flux binaire est décrite en détail dans la documentation de la classe `BufferedIOBase`.

D'autres bibliothèques peuvent fournir des moyens supplémentaires pour créer des flux de texte ou flux binaire. Voir la méthode `socket.socket.makefile()` par exemple.

Raw I/O

Raw I/O (also called *unbuffered I/O*) is generally used as a low-level building-block for binary and text streams ; it is rarely useful to directly manipulate a raw stream from user code. Nevertheless, you can create a raw stream by opening a file in binary mode with buffering disabled :

```
f = open("myfile.jpg", "rb", buffering=0)
```

The raw stream API is described in detail in the docs of `RawIOBase`.

16.2.2 Encodage de texte

The default encoding of `TextIOWrapper` and `open()` is locale-specific (`locale.getencoding()`).

However, many developers forget to specify the encoding when opening text files encoded in UTF-8 (e.g. JSON, TOML, Markdown, etc...) since most Unix platforms use UTF-8 locale by default. This causes bugs because the locale encoding is not UTF-8 for most Windows users. For example :

```
# May not work on Windows when non-ASCII characters in the file.
with open("README.md") as f:
    long_description = f.read()
```

Accordingly, it is highly recommended that you specify the encoding explicitly when opening text files. If you want to use UTF-8, pass `encoding="utf-8"`. To use the current locale encoding, `encoding="locale"` is supported since Python 3.10.

Voir aussi

Le mode UTF-8 de Python
Le mode UTF-8 de Python peut être utilisé pour changer l'encodage par défaut en UTF-8 à partir d'un encodage local spécifique.

PEP 686
Le mode UTF-8 de Python sera par mis par défaut sur Python 3.15.

Opt-in EncodingWarning

Ajouté dans la version 3.10 : See **PEP 597** for more details.

To find where the default locale encoding is used, you can enable the `-X warn_default_encoding` command line option or set the `PYTHONWARNDEFAULTENCODING` environment variable, which will emit an `EncodingWarning` when the default encoding is used.

If you are providing an API that uses `open()` or `TextIOWrapper` and passes `encoding=None` as a parameter, you can use `text_encoding()` so that callers of the API will emit an `EncodingWarning` if they don't pass an `encoding`. However, please consider using UTF-8 by default (i.e. `encoding="utf-8"`) for new APIs.

16.2.3 Interface de haut niveau du module

`io.`**`DEFAULT_BUFFER_SIZE`**

An int containing the default buffer size used by the module's buffered I/O classes. `open()` uses the file's blksize (as obtained by `os.stat()`) if possible.

`io.`**`open`**(*file*, *mode='r'*, *buffering=-1*, *encoding=None*, *errors=None*, *newline=None*, *closefd=True*, *opener=None*)

This is an alias for the builtin `open()` function.

This function raises an *auditing event* `open` with arguments *path*, *mode* and *flags*. The *mode* and *flags* arguments may have been modified or inferred from the original call.

`io.`**`open_code`**(*path*)

Opens the provided file with mode `'rb'`. This function should be used when the intent is to treat the contents as executable code.

path should be a `str` and an absolute path.

The behavior of this function may be overridden by an earlier call to the `PyFile_SetOpenCodeHook()`. However, assuming that *path* is a `str` and an absolute path, `open_code(path)` should always behave the same as `open(path, 'rb')`. Overriding the behavior is intended for additional validation or preprocessing of the file.

Ajouté dans la version 3.8.

`io.`**`text_encoding`**(*encoding*, *stacklevel=2*, */*)

This is a helper function for callables that use `open()` or `TextIOWrapper` and have an `encoding=None` parameter.

This function returns *encoding* if it is not `None`. Otherwise, it returns `"locale"` or `"utf-8"` depending on *UTF-8 Mode*.

This function emits an `EncodingWarning` if `sys.flags.warn_default_encoding` is true and *encoding* is `None`. *stacklevel* specifies where the warning is emitted. For example :

```
def read_text(path, encoding=None):
    encoding = io.text_encoding(encoding)  # stacklevel=2
    with open(path, encoding) as f:
        return f.read()
```

In this example, an `EncodingWarning` is emitted for the caller of `read_text()`.

Voir *Encodage de texte* pour plus d'informations.

Ajouté dans la version 3.10.

Modifié dans la version 3.11 : `text_encoding()` returns "utf-8" when UTF-8 mode is enabled and *encoding* is `None`.

`exception` `io.`**`BlockingIOError`**

This is a compatibility alias for the builtin `BlockingIOError` exception.

exception io.UnsupportedOperation

An exception inheriting *OSError* and *ValueError* that is raised when an unsupported operation is called on a stream.

Voir aussi

sys

contains the standard IO streams : *sys.stdin*, *sys.stdout*, and *sys.stderr*.

16.2.4 Class hierarchy

The implementation of I/O streams is organized as a hierarchy of classes. First *abstract base classes* (ABCs), which are used to specify the various categories of streams, then concrete classes providing the standard stream implementations.

Note

The abstract base classes also provide default implementations of some methods in order to help implementation of concrete stream classes. For example, *BufferedIOBase* provides unoptimized implementations of `readinto()` and `readline()`.

At the top of the I/O hierarchy is the abstract base class *IOBase*. It defines the basic interface to a stream. Note, however, that there is no separation between reading and writing to streams ; implementations are allowed to raise *UnsupportedOperation* if they do not support a given operation.

The *RawIOBase* ABC extends *IOBase*. It deals with the reading and writing of bytes to a stream. *FileIO* subclasses *RawIOBase* to provide an interface to files in the machine's file system.

The *BufferedIOBase* ABC extends *IOBase*. It deals with buffering on a raw binary stream (*RawIOBase*). Its subclasses, *BufferedWriter*, *BufferedReader*, and *BufferedRWPair* buffer raw binary streams that are writable, readable, and both readable and writable, respectively. *BufferedRandom* provides a buffered interface to seekable streams. Another *BufferedIOBase* subclass, *BytesIO*, is a stream of in-memory bytes.

The *TextIOBase* ABC extends *IOBase*. It deals with streams whose bytes represent text, and handles encoding and decoding to and from strings. *TextIOWrapper*, which extends *TextIOBase*, is a buffered text interface to a buffered raw stream (*BufferedIOBase*). Finally, *StringIO* is an in-memory stream for text.

Argument names are not part of the specification, and only the arguments of *open()* are intended to be used as keyword arguments.

The following table summarizes the ABCs provided by the *io* module :

ABC	Inherits	Stub Methods	Méthodes et propriétés de Mixin
IOBase		`fileno`, `seek`, et `truncate`	`close`, `closed`, `__enter__`, `__exit__`, `flush`, `isatty`, `__iter__`, `__next__`, `readable`, `readline`, `readlines`, `seekable`, `tell`, `writable`, and `writelines`
RawIOBase	*IOBase*	`readinto` et `write`	Inherited *IOBase* methods, `read`, and `readall`
BufferedIOBase	*IOBase*	`detach`, `read`, `read1`, et `write`	Inherited *IOBase* methods, `readinto`, and `readinto1`
TextIOBase	*IOBase*	`detach`, `read`, `readline`, et `write`	Inherited *IOBase* methods, `encoding`, `errors`, and `newlines`

I/O Base Classes

class io.IOBase

The abstract base class for all I/O classes.

This class provides empty abstract implementations for many methods that derived classes can override selectively ; the default implementations represent a file that cannot be read, written or seeked.

Even though `IOBase` does not declare `read()` or `write()` because their signatures will vary, implementations and clients should consider those methods part of the interface. Also, implementations may raise a `ValueError` (or `UnsupportedOperation`) when operations they do not support are called.

The basic type used for binary data read from or written to a file is `bytes`. Other *bytes-like objects* are accepted as method arguments too. Text I/O classes work with `str` data.

Note that calling any method (even inquiries) on a closed stream is undefined. Implementations may raise `ValueError` in this case.

`IOBase` (and its subclasses) supports the iterator protocol, meaning that an `IOBase` object can be iterated over yielding the lines in a stream. Lines are defined slightly differently depending on whether the stream is a binary stream (yielding bytes), or a text stream (yielding character strings). See `readline()` below.

`IOBase` is also a context manager and therefore supports the `with` statement. In this example, *file* is closed after the `with` statement's suite is finished---even if an exception occurs :

```
with open('spam.txt', 'w') as file:
    file.write('Spam and eggs!')
```

`IOBase` provides these data attributes and methods :

close()

Flush and close this stream. This method has no effect if the file is already closed. Once the file is closed, any operation on the file (e.g. reading or writing) will raise a `ValueError`.

As a convenience, it is allowed to call this method more than once ; only the first call, however, will have an effect.

closed

`True` if the stream is closed.

fileno()

Return the underlying file descriptor (an integer) of the stream if it exists. An `OSError` is raised if the IO object does not use a file descriptor.

flush()

Flush the write buffers of the stream if applicable. This does nothing for read-only and non-blocking streams.

isatty()

Return `True` if the stream is interactive (i.e., connected to a terminal/tty device).

readable()

Return `True` if the stream can be read from. If `False`, `read()` will raise `OSError`.

readline(*size=-1*, /)

Read and return one line from the stream. If *size* is specified, at most *size* bytes will be read.

The line terminator is always `b'\n'` for binary files ; for text files, the *newline* argument to `open()` can be used to select the line terminator(s) recognized.

readlines(*hint=-1*, /)

Read and return a list of lines from the stream. *hint* can be specified to control the number of lines read : no more lines will be read if the total size (in bytes/characters) of all lines so far exceeds *hint*.

hint values of `0` or less, as well as `None`, are treated as no hint.

Note that it's already possible to iterate on file objects using `for line in file: ...` without calling `file.readlines()`.

seek(*offset*, *whence=os.SEEK_SET*, /)

Change the stream position to the given byte *offset*, interpreted relative to the position indicated by *whence*, and return the new absolute position. Values for *whence* are :
— `os.SEEK_SET` or `0` -- start of the stream (the default) ; *offset* should be zero or positive
— `os.SEEK_CUR` or `1` -- current stream position ; *offset* may be negative
— `os.SEEK_END` or `2` -- end of the stream ; *offset* is usually negative
Ajouté dans la version 3.1 : The `SEEK_*` constants.

Ajouté dans la version 3.3 : Some operating systems could support additional values, like `os.SEEK_HOLE` or `os.SEEK_DATA`. The valid values for a file could depend on it being open in text or binary mode.

seekable()

Return `True` if the stream supports random access. If `False`, `seek()`, `tell()` and `truncate()` will raise `OSError`.

tell()

Return the current stream position.

truncate(*size=None*, /)

Resize the stream to the given *size* in bytes (or the current position if *size* is not specified). The current stream position isn't changed. This resizing can extend or reduce the current file size. In case of extension, the contents of the new file area depend on the platform (on most systems, additional bytes are zero-filled). The new file size is returned.

Modifié dans la version 3.5 : Windows will now zero-fill files when extending.

writable()

Return `True` if the stream supports writing. If `False`, `write()` and `truncate()` will raise `OSError`.

writelines(*lines*, /)

Write a list of lines to the stream. Line separators are not added, so it is usual for each of the lines provided to have a line separator at the end.

__del__()

Prepare for object destruction. `IOBase` provides a default implementation of this method that calls the instance's `close()` method.

class io.**RawIOBase**

Base class for raw binary streams. It inherits from `IOBase`.

Raw binary streams typically provide low-level access to an underlying OS device or API, and do not try to encapsulate it in high-level primitives (this functionality is done at a higher-level in buffered binary streams and text streams, described later in this page).

`RawIOBase` provides these methods in addition to those from `IOBase` :

read(*size=-1*, /)

Read up to *size* bytes from the object and return them. As a convenience, if *size* is unspecified or -1, all bytes until EOF are returned. Otherwise, only one system call is ever made. Fewer than *size* bytes may be returned if the operating system call returns fewer than *size* bytes.

If 0 bytes are returned, and *size* was not 0, this indicates end of file. If the object is in non-blocking mode and no bytes are available, `None` is returned.

The default implementation defers to `readall()` and `readinto()`.

readall()

Read and return all the bytes from the stream until EOF, using multiple calls to the stream if necessary.

readinto(*b*, /)

Read bytes into a pre-allocated, writable *bytes-like object* *b*, and return the number of bytes read. For

example, *b* might be a `bytearray`. If the object is in non-blocking mode and no bytes are available, `None` is returned.

write(*b*, /)

Write the given *bytes-like object*, *b*, to the underlying raw stream, and return the number of bytes written. This can be less than the length of *b* in bytes, depending on specifics of the underlying raw stream, and especially if it is in non-blocking mode. `None` is returned if the raw stream is set not to block and no single byte could be readily written to it. The caller may release or mutate *b* after this method returns, so the implementation should only access *b* during the method call.

class io.**BufferedIOBase**

Base class for binary streams that support some kind of buffering. It inherits from `IOBase`.

The main difference with `RawIOBase` is that methods `read()`, `readinto()` and `write()` will try (respectively) to read as much input as requested or to consume all given output, at the expense of making perhaps more than one system call.

In addition, those methods can raise `BlockingIOError` if the underlying raw stream is in non-blocking mode and cannot take or give enough data ; unlike their `RawIOBase` counterparts, they will never return `None`.

Besides, the `read()` method does not have a default implementation that defers to `readinto()`.

A typical `BufferedIOBase` implementation should not inherit from a `RawIOBase` implementation, but wrap one, like `BufferedWriter` and `BufferedReader` do.

`BufferedIOBase` provides or overrides these data attributes and methods in addition to those from `IOBase` :

raw

The underlying raw stream (a `RawIOBase` instance) that `BufferedIOBase` deals with. This is not part of the `BufferedIOBase` API and may not exist on some implementations.

detach()

Sépare le flux brut sous-jacent du tampon et le renvoie.

After the raw stream has been detached, the buffer is in an unusable state.

Some buffers, like `BytesIO`, do not have the concept of a single raw stream to return from this method. They raise `UnsupportedOperation`.

Ajouté dans la version 3.1.

read(*size=-1*, /)

Read and return up to *size* bytes. If the argument is omitted, `None`, or negative, data is read and returned until EOF is reached. An empty `bytes` object is returned if the stream is already at EOF.

If the argument is positive, and the underlying raw stream is not interactive, multiple raw reads may be issued to satisfy the byte count (unless EOF is reached first). But for interactive raw streams, at most one raw read will be issued, and a short result does not imply that EOF is imminent.

A `BlockingIOError` is raised if the underlying raw stream is in non blocking-mode, and has no data available at the moment.

read1(*size=-1*, /)

Read and return up to *size* bytes, with at most one call to the underlying raw stream's `read()` (or `readinto()`) method. This can be useful if you are implementing your own buffering on top of a `BufferedIOBase` object.

If *size* is `-1` (the default), an arbitrary number of bytes are returned (more than zero unless EOF is reached).

readinto(*b*, /)

Read bytes into a pre-allocated, writable *bytes-like object* *b* and return the number of bytes read. For example, *b* might be a `bytearray`.

Like `read()`, multiple reads may be issued to the underlying raw stream, unless the latter is interactive.

A `BlockingIOError` is raised if the underlying raw stream is in non blocking-mode, and has no data available at the moment.

readinto1(*b*, /)

Read bytes into a pre-allocated, writable *bytes-like object* *b*, using at most one call to the underlying raw stream's `read()` (or `readinto()`) method. Return the number of bytes read.

A `BlockingIOError` is raised if the underlying raw stream is in non blocking-mode, and has no data available at the moment.

Ajouté dans la version 3.5.

write(*b*, /)

Write the given *bytes-like object*, *b*, and return the number of bytes written (always equal to the length of *b* in bytes, since if the write fails an `OSError` will be raised). Depending on the actual implementation, these bytes may be readily written to the underlying stream, or held in a buffer for performance and latency reasons.

When in non-blocking mode, a `BlockingIOError` is raised if the data needed to be written to the raw stream but it couldn't accept all the data without blocking.

The caller may release or mutate *b* after this method returns, so the implementation should only access *b* during the method call.

Raw File I/O

class io.**FileIO**(*name*, *mode='r'*, *closefd=True*, *opener=None*)

A raw binary stream representing an OS-level file containing bytes data. It inherits from `RawIOBase`.

The *name* can be one of two things :

— a character string or `bytes` object representing the path to the file which will be opened. In this case closefd must be `True` (the default) otherwise an error will be raised.

— an integer representing the number of an existing OS-level file descriptor to which the resulting `FileIO` object will give access. When the FileIO object is closed this fd will be closed as well, unless *closefd* is set to `False`.

The *mode* can be `'r'`, `'w'`, `'x'` or `'a'` for reading (default), writing, exclusive creation or appending. The file will be created if it doesn't exist when opened for writing or appending ; it will be truncated when opened for writing. `FileExistsError` will be raised if it already exists when opened for creating. Opening a file for creating implies writing, so this mode behaves in a similar way to `'w'`. Add a `'+'` to the mode to allow simultaneous reading and writing.

The `read()` (when called with a positive argument), `readinto()` and `write()` methods on this class will only make one system call.

Un *opener* personnalisé peut être utilisé en fournissant un appelable à *opener*. Le descripteur de fichier de cet objet fichier sera alors obtenu en appelant *opener* avec (*file*, *flags*). *opener* doit donner un descripteur de fichier ouvert (fournir `os.open` en temps qu'*opener* aura le même effet que donner `None`).

Il n'est *pas possible d'hériter du fichier* nouvellement créé.

See the `open()` built-in function for examples on using the *opener* parameter.

Modifié dans la version 3.3 : The *opener* parameter was added. The `'x'` mode was added.

Modifié dans la version 3.4 : Il n'est plus possible d'hériter de *file*.

`FileIO` provides these data attributes in addition to those from `RawIOBase` and `IOBase` :

mode

The mode as given in the constructor.

name

Le nom du fichier. C'est le descripteur du fichier lorsqu'aucun nom n'est donné dans le constructeur.

Buffered Streams

Buffered I/O streams provide a higher-level interface to an I/O device than raw I/O does.

class io.**BytesIO**(*initial_bytes=b"*)

A binary stream using an in-memory bytes buffer. It inherits from `BufferedIOBase`. The buffer is discarded when the `close()` method is called.

The optional argument *initial_bytes* is a *bytes-like object* that contains initial data.

`BytesIO` provides or overrides these methods in addition to those from `BufferedIOBase` and `IOBase` :

getbuffer()

Return a readable and writable view over the contents of the buffer without copying them. Also, mutating the view will transparently update the contents of the buffer :

```
>>> b = io.BytesIO(b"abcdef")
>>> view = b.getbuffer()
>>> view[2:4] = b"56"
>>> b.getvalue()
b'ab56ef'
```

Note

As long as the view exists, the `BytesIO` object cannot be resized or closed.

Ajouté dans la version 3.2.

getvalue()

Return `bytes` containing the entire contents of the buffer.

read1(*size=-1*, /)

In `BytesIO`, this is the same as `read()`.

Modifié dans la version 3.7 : The *size* argument is now optional.

readinto1(*b*, /)

In `BytesIO`, this is the same as `readinto()`.

Ajouté dans la version 3.5.

class io.**BufferedReader**(*raw*, *buffer_size=DEFAULT_BUFFER_SIZE*)

A buffered binary stream providing higher-level access to a readable, non seekable `RawIOBase` raw binary stream. It inherits from `BufferedIOBase`.

When reading data from this object, a larger amount of data may be requested from the underlying raw stream, and kept in an internal buffer. The buffered data can then be returned directly on subsequent reads.

The constructor creates a `BufferedReader` for the given readable *raw* stream and *buffer_size*. If *buffer_size* is omitted, `DEFAULT_BUFFER_SIZE` is used.

`BufferedReader` provides or overrides these methods in addition to those from `BufferedIOBase` and `IOBase` :

peek(*size=0*, /)

Return bytes from the stream without advancing the position. At most one single read on the raw stream is done to satisfy the call. The number of bytes returned may be less or more than requested.

read(*size=-1*, /)

Read and return *size* bytes, or if *size* is not given or negative, until EOF or if the read call would block in non-blocking mode.

> **Note**
>
> When the underlying raw stream is non-blocking, a `BlockingIOError` may be raised if a read operation cannot be completed immediately.

read1(*size=-1*, /)

Read and return up to *size* bytes with only one call on the raw stream. If at least one byte is buffered, only buffered bytes are returned. Otherwise, one raw stream read call is made.

Modifié dans la version 3.7 : The *size* argument is now optional.

> **Note**
>
> When the underlying raw stream is non-blocking, a `BlockingIOError` may be raised if a read operation cannot be completed immediately.

class io.**BufferedWriter**(*raw*, *buffer_size=DEFAULT_BUFFER_SIZE*)

A buffered binary stream providing higher-level access to a writeable, non seekable `RawIOBase` raw binary stream. It inherits from `BufferedIOBase`.

When writing to this object, data is normally placed into an internal buffer. The buffer will be written out to the underlying `RawIOBase` object under various conditions, including :
— when the buffer gets too small for all pending data ;
— when `flush()` is called ;
— when a `seek()` is requested (for `BufferedRandom` objects) ;
— when the `BufferedWriter` object is closed or destroyed.
The constructor creates a `BufferedWriter` for the given writeable *raw* stream. If the *buffer_size* is not given, it defaults to `DEFAULT_BUFFER_SIZE`.

`BufferedWriter` provides or overrides these methods in addition to those from `BufferedIOBase` and `IOBase` :

flush()

Force bytes held in the buffer into the raw stream. A `BlockingIOError` should be raised if the raw stream blocks.

write(*b*, /)

Write the *bytes-like object*, *b*, and return the number of bytes written. When in non-blocking mode, a `BlockingIOError` is raised if the buffer needs to be written out but the raw stream blocks.

class io.**BufferedRandom**(*raw*, *buffer_size=DEFAULT_BUFFER_SIZE*)

A buffered binary stream providing higher-level access to a seekable `RawIOBase` raw binary stream. It inherits from `BufferedReader` and `BufferedWriter`.

The constructor creates a reader and writer for a seekable raw stream, given in the first argument. If the *buffer_size* is omitted it defaults to `DEFAULT_BUFFER_SIZE`.

`BufferedRandom` is capable of anything `BufferedReader` or `BufferedWriter` can do. In addition, `seek()` and `tell()` are guaranteed to be implemented.

class io.**BufferedRWPair**(*reader*, *writer*, *buffer_size=DEFAULT_BUFFER_SIZE*, /)

A buffered binary stream providing higher-level access to two non seekable `RawIOBase` raw binary streams--one readable, the other writeable. It inherits from `BufferedIOBase`.

reader and *writer* are `RawIOBase` objects that are readable and writeable respectively. If the *buffer_size* is omitted it defaults to `DEFAULT_BUFFER_SIZE`.

`BufferedRWPair` implements all of `BufferedIOBase`'s methods except for `detach()`, which raises `UnsupportedOperation`.

> **Avertissement**
>
> `BufferedRWPair` does not attempt to synchronize accesses to its underlying raw streams. You should not pass it the same object as reader and writer ; use `BufferedRandom` instead.

Entrée/sortie de texte

class io.TextIOBase

Base class for text streams. This class provides a character and line based interface to stream I/O. It inherits from `IOBase`.

`TextIOBase` provides or overrides these data attributes and methods in addition to those from `IOBase` :

encoding

The name of the encoding used to decode the stream's bytes into strings, and to encode strings into bytes.

errors

The error setting of the decoder or encoder.

newlines

A string, a tuple of strings, or `None`, indicating the newlines translated so far. Depending on the implementation and the initial constructor flags, this may not be available.

buffer

The underlying binary buffer (a `BufferedIOBase` instance) that `TextIOBase` deals with. This is not part of the `TextIOBase` API and may not exist in some implementations.

detach()

Separate the underlying binary buffer from the `TextIOBase` and return it.

After the underlying buffer has been detached, the `TextIOBase` is in an unusable state.

Some `TextIOBase` implementations, like `StringIO`, may not have the concept of an underlying buffer and calling this method will raise `UnsupportedOperation`.

Ajouté dans la version 3.1.

read(*size=-1*, /)

Read and return at most *size* characters from the stream as a single `str`. If *size* is negative or `None`, reads until EOF.

readline(*size=-1*, /)

Read until newline or EOF and return a single `str`. If the stream is already at EOF, an empty string is returned.

If *size* is specified, at most *size* characters will be read.

seek(*offset*, *whence=SEEK_SET*, /)

Change the stream position to the given *offset*. Behaviour depends on the *whence* parameter. The default value for *whence* is `SEEK_SET`.

— `SEEK_SET` or `0` : seek from the start of the stream (the default) ; *offset* must either be a number returned by `TextIOBase.tell()`, or zero. Any other *offset* value produces undefined behaviour.

— `SEEK_CUR` or `1` : "seek" to the current position ; *offset* must be zero, which is a no-operation (all other values are unsupported).

— `SEEK_END` or `2` : seek to the end of the stream ; *offset* must be zero (all other values are unsupported).

Return the new absolute position as an opaque number.

Ajouté dans la version 3.1 : The `SEEK_*` constants.

tell()

Return the current stream position as an opaque number. The number does not usually represent a number of bytes in the underlying binary storage.

write(*s*, /)

Write the string *s* to the stream and return the number of characters written.

class io.**TextIOWrapper**(*buffer*, *encoding=None*, *errors=None*, *newline=None*, *line_buffering=False*, *write_through=False*)

A buffered text stream providing higher-level access to a `BufferedIOBase` buffered binary stream. It inherits from `TextIOBase`.

encoding gives the name of the encoding that the stream will be decoded or encoded with. It defaults to `locale.getencoding()`. `encoding="locale"` can be used to specify the current locale's encoding explicitly. See *Encodage de texte* for more information.

errors is an optional string that specifies how encoding and decoding errors are to be handled. Pass `'strict'` to raise a `ValueError` exception if there is an encoding error (the default of `None` has the same effect), or pass `'ignore'` to ignore errors. (Note that ignoring encoding errors can lead to data loss.) `'replace'` causes a replacement marker (such as `'?'`) to be inserted where there is malformed data. `'backslashreplace'` causes malformed data to be replaced by a backslashed escape sequence. When writing, `'xmlcharrefreplace'` (replace with the appropriate XML character reference) or `'namereplace'` (replace with `\N{...}` escape sequences) can be used. Any other error handling name that has been registered with `codecs.register_error()` is also valid.

newline controls how line endings are handled. It can be `None`, `''`, `'\n'`, `'\r'`, and `'\r\n'`. It works as follows :

— When reading input from the stream, if *newline* is `None`, *universal newlines* mode is enabled. Lines in the input can end in `'\n'`, `'\r'`, or `'\r\n'`, and these are translated into `'\n'` before being returned to the caller. If *newline* is `''`, universal newlines mode is enabled, but line endings are returned to the caller untranslated. If *newline* has any of the other legal values, input lines are only terminated by the given string, and the line ending is returned to the caller untranslated.

— Lors de l'écriture, si *newline* est `None`, chaque `'\n'` est remplacé par le séparateur de lignes par défaut du système `os.linesep`. Si *newline* est `*` ou `'\n'` aucun remplacent n'est effectué. Si *newline* est un autre caractère valide, chaque `'\n'` sera remplacé par la chaîne donnée.

If *line_buffering* is `True`, `flush()` is implied when a call to write contains a newline character or a carriage return.

If *write_through* is `True`, calls to `write()` are guaranteed not to be buffered : any data written on the `TextIOWrapper` object is immediately handled to its underlying binary *buffer*.

Modifié dans la version 3.3 : Le paramètre *write_through* a été ajouté.

Modifié dans la version 3.3 : The default *encoding* is now `locale.getpreferredencoding(False)` instead of `locale.getpreferredencoding()`. Don't change temporary the locale encoding using `locale.setlocale()`, use the current locale encoding instead of the user preferred encoding.

Modifié dans la version 3.10 : The *encoding* argument now supports the `"locale"` dummy encoding name.

> **Note**
>
> When the underlying raw stream is non-blocking, a `BlockingIOError` may be raised if a read operation cannot be completed immediately.

`TextIOWrapper` provides these data attributes and methods in addition to those from `TextIOBase` and `IOBase` :

line_buffering

Whether line buffering is enabled.

write_through

Whether writes are passed immediately to the underlying binary buffer.

Ajouté dans la version 3.7.

reconfigure(*, *encoding=None*, *errors=None*, *newline=None*, *line_buffering=None*, *write_through=None*)

Reconfigure this text stream using new settings for *encoding*, *errors*, *newline*, *line_buffering* and *write_through*.

Parameters not specified keep current settings, except `errors='strict'` is used when *encoding* is specified but *errors* is not specified.

Il n'est pas possible de modifier l'encodage ou une nouvelle ligne si des données ont déjà été lues à partir du flux. En revanche, il est possible de modifier l'encodage après l'écriture.

Cette méthode effectue un nettoyage implicite du flux avant de définir les nouveaux paramètres.

Ajouté dans la version 3.7.

Modifié dans la version 3.11 : La méthode prend en charge l'option `encoding="locale"`.

seek(*cookie*, *whence=os.SEEK_SET*, /)

Set the stream position. Return the new stream position as an `int`.

Four operations are supported, given by the following argument combinations :
— `seek(0, SEEK_SET)` : Rewind to the start of the stream.
— `seek(cookie, SEEK_SET)` : Restore a previous position ; *cookie* **must be** a number returned by `tell()`.
— `seek(0, SEEK_END)` : Fast-forward to the end of the stream.
— `seek(0, SEEK_CUR)` : Leave the current stream position unchanged.
Any other argument combinations are invalid, and may raise exceptions.

Voir aussi

`os.SEEK_SET`, `os.SEEK_CUR`, and `os.SEEK_END`.

tell()

Return the stream position as an opaque number. The return value of `tell()` can be given as input to `seek()`, to restore a previous stream position.

class io.**StringIO**(*initial_value=''*, *newline='\n'*)

A text stream using an in-memory text buffer. It inherits from `TextIOBase`.

The text buffer is discarded when the `close()` method is called.

The initial value of the buffer can be set by providing *initial_value*. If newline translation is enabled, newlines will be encoded as if by `write()`. The stream is positioned at the start of the buffer which emulates opening an existing file in a `w+` mode, making it ready for an immediate write from the beginning or for a write that would overwrite the initial value. To emulate opening a file in an `a+` mode ready for appending, use `f.seek(0, io.SEEK_END)` to reposition the stream at the end of the buffer.

The *newline* argument works like that of `TextIOWrapper`, except that when writing output to the stream, if *newline* is `None`, newlines are written as `\n` on all platforms.

`StringIO` provides this method in addition to those from `TextIOBase` and `IOBase` :

getvalue()

Return a `str` containing the entire contents of the buffer. Newlines are decoded as if by `read()`, although the stream position is not changed.

Exemple d'utilisation :

```
import io

output = io.StringIO()
output.write('First line.\n')
print('Second line.', file=output)

# Retrieve file contents -- this will be
```

(suite sur la page suivante)

(suite de la page précédente)

```
# 'First line.\nSecond line.\n'
contents = output.getvalue()

# Close object and discard memory buffer --
# .getvalue() will now raise an exception.
output.close()
```

class io.IncrementalNewlineDecoder

A helper codec that decodes newlines for *universal newlines* mode. It inherits from `codecs.IncrementalDecoder`.

16.2.5 Performances

Cette section aborde les performances des implémentations concrètes d'Entrée/Sortie fournies.

Binary I/O

By reading and writing only large chunks of data even when the user asks for a single byte, buffered I/O hides any inefficiency in calling and executing the operating system's unbuffered I/O routines. The gain depends on the OS and the kind of I/O which is performed. For example, on some modern OSes such as Linux, unbuffered disk I/O can be as fast as buffered I/O. The bottom line, however, is that buffered I/O offers predictable performance regardless of the platform and the backing device. Therefore, it is almost always preferable to use buffered I/O rather than unbuffered I/O for binary data.

Entrée/sortie de texte

Text I/O over a binary storage (such as a file) is significantly slower than binary I/O over the same storage, because it requires conversions between unicode and binary data using a character codec. This can become noticeable handling huge amounts of text data like large log files. Also, `tell()` and `seek()` are both quite slow due to the reconstruction algorithm used.

`StringIO`, however, is a native in-memory unicode container and will exhibit similar speed to `BytesIO`.

Fils d'exécution

`FileIO` objects are thread-safe to the extent that the operating system calls (such as `read(2)` under Unix) they wrap are thread-safe too.

Binary buffered objects (instances of `BufferedReader`, `BufferedWriter`, `BufferedRandom` and `BufferedRWPair`) protect their internal structures using a lock; it is therefore safe to call them from multiple threads at once.

les objets `TextIOWrapper` ne sont pas compatibles avec les programmes à fils d'exécutions multiples.

Réentrance

Binary buffered objects (instances of `BufferedReader`, `BufferedWriter`, `BufferedRandom` and `BufferedRWPair`) are not reentrant. While reentrant calls will not happen in normal situations, they can arise from doing I/O in a `signal` handler. If a thread tries to re-enter a buffered object which it is already accessing, a `RuntimeError` is raised. Note this doesn't prohibit a different thread from entering the buffered object.

The above implicitly extends to text files, since the `open()` function will wrap a buffered object inside a `TextIOWrapper`. This includes standard streams and therefore affects the built-in `print()` function as well.

16.3 `time` --- Time access and conversions

Ce module fournit différentes fonctions liées au temps. Pour les fonctionnalités associées, voir aussi les modules `datetime` et `calendar`.

Bien que ce module soit toujours disponible, toutes les fonctions ne sont pas disponibles sur toutes les plateformes. La plupart des fonctions définies dans ce module délèguent à des fonctions de même nom de la bibliothèque C. Il peut parfois être utile de consulter la documentation de la plate-forme, car la sémantique de ces fonctions peut varier.

Vous trouvez ci-dessous, mises en ordre, quelques explications relative à la terminologie et aux conventions.

— L'*epoch* est le point de départ du temps, le résultat de `time.gmtime(0)` est le 1er janvier 1970 à 00 :00 :00 (UTC) pour toutes les plateformes.

— Le terme *secondes depuis *epoch** désigne le nombre total de secondes écoulées depuis *epoch*, souvent en excluant les secondes intercalaires (leap seconds). Les secondes intercalaires sont exclues de ce total sur toutes les plates-formes conformes POSIX.

— Les fonctions de ce module peuvent ne pas gérer les dates et heures antérieures à *epoch* ou dans un avenir lointain. Le seuil du futur est déterminé par la bibliothèque C ; pour les systèmes 32 bits, il s'agit généralement de 2038.

— La fonction `strptime()` peut analyser des années à 2 chiffres lorsque le format `%y` est spécifié. Lorsque les années à deux chiffres sont analysées, elles sont converties conformément aux normes POSIX et ISO C : les valeurs 69—99 correspondent à 1969—1999 et les valeurs 0—68 à 2000—2068.

— UTC désigne le temps universel coordonné (*Coordinated Universal Time* en anglais), anciennement l'heure de Greenwich (ou GMT). L'acronyme UTC n'est pas une erreur mais un compromis entre l'anglais et le français.

— Le DST (*Daylight Saving Time*) correspond à l'heure d'été, un ajustement du fuseau horaire d'une heure (généralement) pendant une partie de l'année. Les règles de DST sont magiques (déterminées par la loi locale) et peuvent changer d'année en année. La bibliothèque C possède une table contenant les règles locales (souvent, elle est lue dans un fichier système par souci de souplesse) et constitue la seule source fiable.

— La précision des diverses fonctions en temps réel peut être inférieure à celle suggérée par les unités dans lesquelles leur valeur ou leur argument est exprimé. Par exemple, sur la plupart des systèmes Unix, l'horloge ne « bat » que 50 ou 100 fois par seconde.

— On the other hand, the precision of `time()` and `sleep()` is better than their Unix equivalents : times are expressed as floating-point numbers, `time()` returns the most accurate time available (using Unix `gettimeofday()` where available), and `sleep()` will accept a time with a nonzero fraction (Unix `select()` is used to implement this, where available).

— La valeur temporelle renvoyée par `gmtime()`, `localtime()` et `strptime()`, et acceptée par `asctime()`, `mktime()` et `strftime()`, est une séquence de 9 nombres entiers. Les valeurs de retour de `gmtime()`, `localtime()` et `strptime()` proposent également des noms d'attributs pour des champs individuels.

  Voir `struct_time` pour une description de ces objets.

  Modifié dans la version 3.3 : The `struct_time` type was extended to provide the `tm_gmtoff` and `tm_zone` attributes when platform supports corresponding `struct tm` members.

  Modifié dans la version 3.6 : The `struct_time` attributes `tm_gmtoff` and `tm_zone` are now available on all platforms.

— Utilisez les fonctions suivantes pour convertir des représentations temporelles :

De	À	Utilisez
secondes depuis *epoch*	`struct_time` en UTC	`gmtime()`
secondes depuis *epoch*	`struct_time` en heure locale	`localtime()`
`struct_time` en UTC	secondes depuis *epoch*	`calendar.timegm()`
`struct_time` en heure locale	secondes depuis *epoch*	`mktime()`

16.3.1 Fonctions

`time.`**`asctime`**`(`[*t*]`)`

Convertit un *n*-uplet ou `struct_time` représentant une heure renvoyée par `gmtime()` ou `localtime()` en une chaîne de la forme suivante : `'Sun Jun 20 23:21:05 1993'`. Le numéro du jour est un champ de deux caractères complété par une espace si celui-ci n'a qu'un seul chiffre, par exemple : `'Wed Jun 9 04:26:40 1993'`.

Si *t* n'est pas fourni, l'heure actuelle renvoyée par `localtime()` est utilisée. Les informations sur les paramètres régionaux ne sont pas utilisées par `asctime()`.

> **Note**
>
> Contrairement à la fonction C du même nom, `asctime()` n'ajoute pas de caractère de fin de ligne.

`time.`**`pthread_getcpuclockid`**(*thread_id*)

Renvoie le *clk_id* de l'horloge du temps CPU spécifique au fil d'exécution pour le *thread_id* spécifié.

Utilisez `threading.get_ident()` ou l'attribut `ident` de `threading.Thread` pour obtenir une valeur appropriée pour *thread_id*.

> **Avertissement**
>
> Passer un *thread_id* invalide ou arrivé à expiration peut entraîner un comportement indéfini, tel qu'une erreur de segmentation.

Availability : Unix

Se reporter à la page de manuel `pthread_getcpuclockid(3)` pour plus d'informations.

Ajouté dans la version 3.7.

`time.`**`clock_getres`**(*clk_id*)

Renvoie la résolution (précision) de l'horloge *clk_id*. Référez-vous à *Constantes d'identification d'horloge* pour une liste des valeurs acceptées pour *clk_id*.

Availability : Unix.

Ajouté dans la version 3.3.

`time.`**`clock_gettime`**(*clk_id*) → *float*

Renvoie l'heure de l'horloge *clk_id*. Référez-vous à *Constantes d'identification d'horloge* pour une liste des valeurs acceptées pour *clk_id*.

Use `clock_gettime_ns()` to avoid the precision loss caused by the `float` type.

Availability : Unix.

Ajouté dans la version 3.3.

`time.`**`clock_gettime_ns`**(*clk_id*) → *int*

Similaire à `clock_gettime()` mais le temps renvoyé est exprimé en nanosecondes.

Availability : Unix.

Ajouté dans la version 3.7.

`time.`**`clock_settime`**(*clk_id*, *time* : float)

Définit l'heure de l'horloge *clk_id*. Actuellement, `CLOCK_REALTIME` est la seule valeur acceptée pour *clk_id*.

Use `clock_settime_ns()` to avoid the precision loss caused by the `float` type.

Availability : Unix, not Android, not iOS.

Ajouté dans la version 3.3.

`time.`**`clock_settime_ns`**(*clk_id*, *time* : int)

Similaire à `clock_settime()` mais définit l'heure avec des nanosecondes.

Availability : Unix, not Android, not iOS.

Ajouté dans la version 3.7.

`time.ctime([secs])`

Convertit une heure exprimée en secondes depuis *epoch* en une chaîne représentant l'heure locale sous la forme suivante : `'Sun Jun 20 23:21:05 1993'`. Le numéro du jour est un champ de deux caractères complété par une espace si celui-ci n'a qu'un seul chiffre, par exemple : `'Wed Jun 9 04:26:40 1993'`.

Si *secs* n'est pas fourni ou vaut `None`, l'heure actuelle renvoyée par `time()` est utilisée. `ctime(secs)` est équivalent à `asctime(localtime(secs))`. Les informations sur les paramètres régionaux ne sont pas utilisées par `ctime()`.

`time.get_clock_info(name)`

Renvoie des informations sur l'horloge spécifiée en tant qu'objet d'espace de nom. Les noms d'horloge pris en charge et les fonctions correspondantes permettant de lire leur valeur sont les suivants :

— `'monotonic'` : `time.monotonic()`
— `'perf_counter'` : `time.perf_counter()`
— `'process_time'` : `time.process_time()`
— `'thread_time'` : `time.thread_time()`
— `'time'` : `time.time()`

Le résultat a les attributs suivants :

— *adjustable* : `True` si l'horloge peut être changée automatiquement (par exemple par un démon NTP) ou manuellement par l'administrateur système, `False` autrement
— *implementation* : nom de la fonction C sous-jacente utilisée pour obtenir la valeur d'horloge. Voir *Constantes d'identification d'horloge* pour les valeurs possibles.
— *monotonic* : `True` si l'horloge ne peut pas revenir en arrière, `False` autrement
— *resolution* : La résolution de l'horloge en secondes (`float`)

Ajouté dans la version 3.3.

`time.gmtime([secs])`

Convertit un temps exprimé en secondes depuis *epoch* en un `struct_time` au format UTC dans lequel le drapeau *dst* est toujours égal à zéro. Si *secs* n'est pas fourni ou vaut `None`, l'heure actuelle renvoyée par `time()` est utilisée. Les fractions de seconde sont ignorées. Voir ci-dessus pour une description de l'objet `struct_time`. Voir `calendar.timegm()` pour l'inverse de cette fonction.

`time.localtime([secs])`

Comme `gmtime()` mais convertit le résultat en heure locale. Si *secs* n'est pas fourni ou vaut `None`, l'heure actuelle renvoyée par `time()` est utilisée. Le drapeau *dst* est mis à `1` lorsque l'heure d'été s'applique à l'heure indiquée.

`localtime()` may raise `OverflowError`, if the timestamp is outside the range of values supported by the platform C `localtime()` or `gmtime()` functions, and `OSError` on `localtime()` or `gmtime()` failure. It's common for this to be restricted to years between 1970 and 2038.

`time.mktime(t)`

This is the inverse function of `localtime()`. Its argument is the `struct_time` or full 9-tuple (since the dst flag is needed ; use `-1` as the dst flag if it is unknown) which expresses the time in *local* time, not UTC. It returns a floating-point number, for compatibility with `time()`. If the input value cannot be represented as a valid time, either `OverflowError` or `ValueError` will be raised (which depends on whether the invalid value is caught by Python or the underlying C libraries). The earliest date for which it can generate a time is platform-dependent.

`time.monotonic() → float`

Renvoie la valeur (en quelques fractions de secondes) d'une horloge monotone, c'est-à-dire une horloge qui ne peut pas revenir en arrière. L'horloge n'est pas affectée par les mises à jour de l'horloge système. Le point de référence de la valeur renvoyée n'est pas défini, de sorte que seule la différence entre les résultats d'appels consécutifs est valide.

Clock :

— On Windows, call `QueryPerformanceCounter()` and `QueryPerformanceFrequency()`.
— On macOS, call `mach_absolute_time()` and `mach_timebase_info()`.
— On HP-UX, call `gethrtime()`.
— Call `clock_gettime(CLOCK_HIGHRES)` if available.
— Otherwise, call `clock_gettime(CLOCK_MONOTONIC)`.

Use `monotonic_ns()` to avoid the precision loss caused by the `float` type.

Ajouté dans la version 3.3.

Modifié dans la version 3.5 : La fonction est maintenant toujours disponible et toujours à l'échelle du système.

Modifié dans la version 3.10 : Sur macOS, la fonction est maintenant toujours à l'échelle du système.

`time.`**`monotonic_ns`**`()` → *int*

Similaire à `monotonique()`, mais le temps de retour est exprimé en nanosecondes.

Ajouté dans la version 3.7.

`time.`**`perf_counter`**`()` → *float*

Renvoie la valeur (en quelques fractions de secondes) d'un compteur de performance, c'est-à-dire une horloge avec la résolution disponible la plus élevée possible pour mesurer une courte durée. Cela inclut le temps écoulé pendant le sommeil et concerne l'ensemble du système. Le point de référence de la valeur renvoyée n'est pas défini, de sorte que seule la différence entre les résultats d'appels consécutifs est valide.

Particularité de l'implémentation CPython : On CPython, use the same clock as `time.monotonic()` and is a monotonic clock, i.e. a clock that cannot go backwards.

Use `perf_counter_ns()` to avoid the precision loss caused by the `float` type.

Ajouté dans la version 3.3.

Modifié dans la version 3.10 : On Windows, the function is now system-wide.

Modifié dans la version 3.13 : Use the same clock as `time.monotonic()`.

`time.`**`perf_counter_ns`**`()` → *int*

Similaire à `perf_counter()`, mais renvoie le temps en nanosecondes.

Ajouté dans la version 3.7.

`time.`**`process_time`**`()` → *float*

Return the value (in fractional seconds) of the sum of the system and user CPU time of the current process. It does not include time elapsed during sleep. It is process-wide by definition. The reference point of the returned value is undefined, so that only the difference between the results of two calls is valid.

Use `process_time_ns()` to avoid the precision loss caused by the `float` type.

Ajouté dans la version 3.3.

`time.`**`process_time_ns`**`()` → *int*

Similaire à `process_time()` mais renvoie le temps en nanosecondes.

Ajouté dans la version 3.7.

`time.`**`sleep`**`(`*secs*`)`

Suspend execution of the calling thread for the given number of seconds. The argument may be a floating-point number to indicate a more precise sleep time.

If the sleep is interrupted by a signal and no exception is raised by the signal handler, the sleep is restarted with a recomputed timeout.

The suspension time may be longer than requested by an arbitrary amount, because of the scheduling of other activity in the system.

Windows implementation

On Windows, if *secs* is zero, the thread relinquishes the remainder of its time slice to any other thread that is ready to run. If there are no other threads ready to run, the function returns immediately, and the thread continues execution. On Windows 8.1 and newer the implementation uses a high-resolution timer which provides resolution of 100 nanoseconds. If *secs* is zero, `Sleep(0)` is used.

Unix implementation

— Use `clock_nanosleep()` if available (resolution : 1 nanosecond) ;
— Or use `nanosleep()` if available (resolution : 1 nanosecond) ;
— Or use `select()` (resolution : 1 microsecond).

> **Note**
>
> To emulate a "no-op", use `pass` instead of `time.sleep(0)`.
>
> To voluntarily relinquish the CPU, specify a real-time *scheduling policy* and use `os.sched_yield()` instead.

Raises an *auditing event* `time.sleep` with argument `secs`.

Modifié dans la version 3.5 : La fonction dort maintenant au moins *secondes* même si le sommeil est interrompu par un signal, sauf si le gestionnaire de signaux lève une exception (voir **PEP 475** pour la justification).

Modifié dans la version 3.11 : On Unix, the `clock_nanosleep()` and `nanosleep()` functions are now used if available. On Windows, a waitable timer is now used.

Modifié dans la version 3.13 : Raises an auditing event.

`time.`**`strftime`**(*format*[, *t*])

Convertit un *n*-uplet ou `struct_time` représentant une heure renvoyée par `gmtime()` ou `localtime()` en une chaîne spécifiée par l'argument *format*. Si *t* n'est pas fourni, l'heure actuelle renvoyée par `localtime()` est utilisée. *format* doit être une chaîne. Si l'un des champs de *t* se situe en dehors de la plage autorisée, une `ValueError` est levée .

0 est un argument légal pour toute position dans le *n*-uplet temporel ; s'il est normalement illégal, la valeur est forcée à une valeur correcte.

Les directives suivantes peuvent être incorporées dans la chaîne *format*. Ils sont affichés sans la spécification facultative de largeur de champ ni de précision, et sont remplacés par les caractères indiqués dans le résultat de `strftime()` :

Directive	Signification	Notes
`%a`	Nom abrégé du jour de la semaine selon les paramètres régionaux.	
`%A`	Le nom de semaine complet de la région.	
`%b`	Nom abrégé du mois de la région.	
`%B`	Nom complet du mois de la région.	
`%c`	Représentation appropriée de la date et de l'heure selon les paramètres régionaux.	
`%d`	Jour du mois sous forme décimale [01,31].	
`%f`	**Microseconds as a decimal number** [000000,999999].	(1)
`%H`	Heure (horloge sur 24 heures) sous forme de nombre décimal [00,23].	
`%I`	Heure (horloge sur 12 heures) sous forme de nombre décimal [01,12].	
`%j`	Jour de l'année sous forme de nombre décimal [001,366].	
`%m`	Mois sous forme décimale [01,12].	
`%M`	Minutes sous forme décimale [00,59].	
`%p`	L'équivalent local de AM ou PM.	(2)
`%S`	Secondes sous forme de nombre décimal [00,61].	(3)
`%U`	Numéro de semaine de l'année (dimanche en tant que premier jour de la semaine) sous forme décimale [00,53]. Tous les jours d'une nouvelle année précédant le premier dimanche sont considérés comme appartenant à la semaine 0.	(4)
`%u`	Day of the week (Monday is 1 ; Sunday is 7) as a decimal number [1, 7].	
`%w`	Jour de la semaine sous forme de nombre décimal [0 (dimanche), 6].	
`%W`	Numéro de semaine de l'année (lundi comme premier jour de la semaine) sous forme décimale [00,53]. Tous les jours d'une nouvelle année précédant le premier lundi sont considérés comme appartenant à la semaine 0.	(4)
`%x`	Représentation de la date appropriée par les paramètres régionaux.	
`%X`	Représentation locale de l'heure.	
`%y`	Année sans siècle comme un nombre décimal [00, 99].	
`%Y`	Année complète sur quatre chiffres.	
`%z`	Décalage de fuseau horaire indi-	

Notes :

(1) The `%f` format directive only applies to `strptime()`, not to `strftime()`. However, see also `datetime.datetime.strptime()` and `datetime.datetime.strftime()` where the `%f` format directive *applies to microseconds*.

(2) Lorsqu'elle est utilisée avec la fonction `strptime()`, la directive `%p` n'affecte le champ d'heure en sortie que si la directive `%I` est utilisée pour analyser l'heure.

(3) La plage est en réalité de `0` à `61` ; la valeur `60` est valide dans les *timestamps* représentant des secondes intercalaires (leap seconds) et la valeur `61` est prise en charge pour des raisons historiques.

(4) Lorsqu'elles sont utilisées avec la fonction `strptime()`, `%U` et `%W` ne sont utilisées que dans les calculs lorsque le jour de la semaine et l'année sont spécifiés.

Voici un exemple de format de date compatible avec celui spécifié dans la norme de courrier électronique Internet suivante **RFC 2822**.[1]

```
>>> from time import gmtime, strftime
>>> strftime("%a, %d %b %Y %H:%M:%S +0000", gmtime())
'Thu, 28 Jun 2001 14:17:15 +0000'
```

Des directives supplémentaires peuvent être prises en charge sur certaines plates-formes, mais seules celles énumérées ici ont une signification normalisée par ANSI C. Pour voir la liste complète des codes de format pris en charge sur votre plate-forme, consultez la documentation `strftime(3)`.

Sur certaines plates-formes, une spécification facultative de largeur et de précision de champ peut suivre immédiatement le `'%'` initial d'une directive dans l'ordre suivant ; ce n'est pas non plus portable. La largeur du champ est normalement 2 sauf pour `%j` où il est 3.

`time.`**`strptime`**(*string*[, *format*])

Analyse une chaîne représentant une heure selon un format. La valeur renvoyée est une `struct_time` tel que renvoyé par `gmtime()` ou `localtime()`.

Le paramètre *format* utilise les mêmes directives que celles utilisées par `strftime()` ; la valeur par défaut est `"%a %b %d %H:%M:%S %Y"` qui correspond à la mise en forme renvoyée par `ctime()`. Si *string* ne peut pas être analysé selon *format*, ou s'il contient trop de données après l'analyse, une exception `ValueError` est levée. Les valeurs par défaut utilisées pour renseigner les données manquantes lorsque des valeurs plus précises ne peuvent pas être inférées sont `(1900, 1, 1, 0, 0, 0, 0, 1, -1)`. *string* et *format* doivent être des chaînes.

Par exemple :

```
>>> import time
>>> time.strptime("30 Nov 00", "%d %b %y")
time.struct_time(tm_year=2000, tm_mon=11, tm_mday=30, tm_hour=0, tm_min=0,
                 tm_sec=0, tm_wday=3, tm_yday=335, tm_isdst=-1)
```

La prise en charge de la directive `%Z` est basée sur les valeurs contenues dans `tzname` et sur le fait de savoir si `daylight` est vrai. Pour cette raison, il est spécifique à la plate-forme, à l'exception de la reconnaissance des heures UTC et GMT, qui sont toujours connues (et considérées comme des fuseaux horaires ne respectant pas l'heure d'été).

Seules les directives spécifiées dans la documentation sont prises en charge. Parce que `strftime()` peut être implémenté différemment sur chaque plate-forme, il peut parfois offrir plus de directives que celles listées. Mais `strptime()` est indépendant de toute plate-forme et ne supporte donc pas nécessairement toutes les directives disponibles qui ne sont pas documentées comme gérées.

1. L'utilisation de `%Z` est maintenant obsolète, mais l'échappement `%z` qui donne le décalage horaire jusqu'à la minute et dépendant des paramètres régionaux n'est pas pris en charge par toutes les bibliothèques C ANSI. En outre, une lecture stricte du standard **RFC 822** de 1982 milite pour une année à deux chiffres (%y plutôt que %Y), mais la pratique a migré vers des années à 4 chiffres de long avant l'année 2000. Après cela, la **RFC 822** est devenue obsolète et l'année à 4 chiffres a été recommandée pour la première fois par la **RFC 1123** puis rendue obligatoire par la **RFC 2822**.

class time.**struct_time**

Le type de la séquence de valeur temporelle renvoyé par `gmtime()`, `localtime()` et `strptime()`. Semblable à un *named tuple* : ses valeurs sont accessibles par index et par nom d'attribut. Les valeurs suivantes sont présentes :

Index	Attribut	Valeurs
0	**tm_year**	(par exemple, 1993)
1	**tm_mon**	plage [1, 12]
2	**tm_mday**	plage [1, 31]
3	**tm_hour**	plage [0, 23]
4	**tm_min**	plage [0, 59]
5	**tm_sec**	range [0, 61] ; see *Note (2)* in `strftime()`
6	**tm_wday**	range [0, 6] ; Monday is 0
7	**tm_yday**	plage [1, 366]
8	**tm_isdst**	`0`, `1` or `-1` ; voir en bas
N/A	**tm_zone**	abréviation du nom du fuseau horaire
N/A	**tm_gmtoff**	décalage à l'est de UTC en secondes

Notez que contrairement à la structure C, la valeur du mois est une plage de [1, 12], pas de [0, 11].

Dans les appels à `mktime()`, `tm_isdst` peut être défini sur 1 lorsque l'heure d'été est en vigueur et sur 0 lorsque ce n'est pas le cas. Une valeur de `-1` indique que cela n'est pas connu et entraînera généralement le remplissage de l'état correct.

Lorsqu'un *n*-uplet de longueur incorrecte est passé à une fonction acceptant une `struct_time`, ou comportant des éléments de type incorrect, une exception `TypeError` est levé.

time.**time**() → *float*

Return the time in seconds since the *epoch* as a floating-point number. The handling of leap seconds is platform dependent. On Windows and most Unix systems, the leap seconds are not counted towards the time in seconds since the *epoch*. This is commonly referred to as Unix time.

Note that even though the time is always returned as a floating-point number, not all systems provide time with a better precision than 1 second. While this function normally returns non-decreasing values, it can return a lower value than a previous call if the system clock has been set back between the two calls.

Le nombre renvoyé par `time()` peut être converti en un format d'heure plus courant (année, mois, jour, heure, etc.) en UTC en le transmettant à la fonction `gmtime()` ou dans heure locale en le transmettant à la fonction

`localtime()`. Dans les deux cas, un objet `struct_time` est renvoyé, à partir duquel les composants de la date du calendrier peuvent être consultés en tant qu'attributs.

Clock :
— On Windows, call `GetSystemTimeAsFileTime()`.
— Call `clock_gettime(CLOCK_REALTIME)` if available.
— Otherwise, call `gettimeofday()`.
Use `time_ns()` to avoid the precision loss caused by the `float` type.

`time.time_ns()` → *int*

Similaire à `time()` mais renvoie le nombre de nanosecondes depuis *epoch* sous la forme d'un entier.

Ajouté dans la version 3.7.

`time.thread_time()` → *float*

Renvoie la valeur (en quelques fractions de secondes) de la somme des temps processeur système et utilisateur du fil d'exécution en cours. Il ne comprend pas le temps écoulé pendant le sommeil. Il est spécifique au fil d'exécution par définition. Le point de référence de la valeur renvoyée est indéfini, de sorte que seule la différence entre les résultats d'appels consécutifs dans le même fil d'exécution est valide.

Use `thread_time_ns()` to avoid the precision loss caused by the `float` type.

Availability : Linux, Unix, Windows.

Systèmes prenant en charge `CLOCK_THREAD_CPUTIME_ID`.

Ajouté dans la version 3.7.

`time.thread_time_ns()` → *int*

Similaire à `thread_time()` mais renvoie le temps en nanosecondes.

Ajouté dans la version 3.7.

`time.tzset()`

Réinitialise les règles de conversion de temps utilisées par les routines de la bibliothèque. La variable d'environnement `TZ` spécifie comment cela est effectué. La fonction définira également les variables `tzname` (à partir de la variable d'environnement `TZ`), `timezone` (secondes non DST à l'ouest de l'UTC), `altzone` (secondes DST à l'ouest de UTC) et `daylight` (à 0 si ce fuseau horaire ne comporte aucune règle d'heure d'été, ou non nul s'il existe une heure, passée, présente ou future lorsque l'heure d'été est appliquée).

Availability : Unix.

> **Note**
>
> Bien que dans de nombreux cas, la modification de la variable d'environnement `TZ` puisse affecter la sortie de fonctions telles que `localtime()` sans appeler `tzset()`, ce comportement n'est pas garanti.
>
> La variable d'environnement `TZ` ne doit contenir aucun espace.

Le format standard de la variable d'environnement `TZ` est (espaces ajoutés pour plus de clarté) :

```
std offset [dst [offset [,start[/time], end[/time]]]]
```

Où les composants sont :

`std` et `dst`
Trois alphanumériques ou plus donnant les abréviations du fuseau horaire. Ceux-ci seront propagés dans *time.tzname*

`offset`
Le décalage a la forme suivante : `± hh[:mm[:ss]]`. Cela indique la valeur ajoutée à l'heure locale pour arriver à UTC. S'il est précédé d'un `'-'`, le fuseau horaire est à l'est du Premier Méridien ; sinon, c'est l'ouest. Si aucun décalage ne suit *dst*, l'heure d'été est supposée être en avance d'une heure sur l'heure standard.

start[/time], end[/time]
Indique quand passer à DST et en revenir. Le format des dates de début et de fin est l'un des suivants :

Jn
Le jour Julien *n* (1 <= *n* <= 365). Les jours bissextiles ne sont pas comptabilisés. Par conséquent, le 28 février est le 59e jour et le 1er mars est le 60e jour de toutes les années.

n
Le jour Julien de base zéro (0 <= *n* <= 365). Les jours bissextiles sont comptés et il est possible de se référer au 29 février.

Mm.n.d
Le *d* jour (0 <= *d* <= 6) de la semaine *n* du mois *m* de l'année (1 <= *n* <= 5, 1 <= *m* <= 12, où semaine 5 signifie "le *dernier* jour du mois *m*" pouvant se produire au cours de la quatrième ou de la cinquième semaine). La semaine 1 est la première semaine au cours de laquelle le *jour* se produit. Le jour zéro est un dimanche.

`time` a le même format que `offset` sauf qu'aucun signe de direction (`'-'` ou `'+'`) n'est autorisé. La valeur par défaut, si l'heure n'est pas spécifiée, est 02 :00 :00.

```
>>> os.environ['TZ'] = 'EST+05EDT,M4.1.0,M10.5.0'
>>> time.tzset()
>>> time.strftime('%X %x %Z')
'02:07:36 05/08/03 EDT'
>>> os.environ['TZ'] = 'AEST-10AEDT-11,M10.5.0,M3.5.0'
>>> time.tzset()
>>> time.strftime('%X %x %Z')
'16:08:12 05/08/03 AEST'
```

Sur de nombreux systèmes Unix (y compris *BSD, Linux, Solaris et Darwin), il est plus pratique d'utiliser la base de données *zoneinfo* (`tzfile (5)`) du système pour spécifier les règles de fuseau horaire. Pour ce faire, définissez la variable d'environnement `TZ` sur le chemin du fichier de fuseau horaire requis, par rapport à la racine de la base de données du système *zoneinfo*, généralement situé à `/usr/share/zoneinfo`. Par exemple, `'US/Eastern'`, `'Australia/Melbourne'`, `'Egypt'` ou `'Europe/Amsterdam'`.

```
>>> os.environ['TZ'] = 'US/Eastern'
>>> time.tzset()
>>> time.tzname
('EST', 'EDT')
>>> os.environ['TZ'] = 'Egypt'
>>> time.tzset()
>>> time.tzname
('EET', 'EEST')
```

16.3.2 Constantes d'identification d'horloge

Ces constantes sont utilisées comme paramètres pour `clock_getres()` et `clock_gettime()`.

`time.`**CLOCK_BOOTTIME**

Identique à `CLOCK_MONOTONIC`, sauf qu'elle inclut également toute suspension du système.

Cela permet aux applications d'obtenir une horloge monotone tenant compte de la suspension sans avoir à gérer les complications de `CLOCK_REALTIME`, qui peuvent présenter des discontinuités si l'heure est modifiée à l'aide de `settimeofday()` ou similaire.

Availability : Linux >= 2.6.39.

Ajouté dans la version 3.7.

`time.`**CLOCK_HIGHRES**

Le système d'exploitation Solaris dispose d'une horloge `CLOCK_HIGHRES` qui tente d'utiliser une source matérielle optimale et peut donner une résolution proche de la nanoseconde. `CLOCK_HIGHRES` est l'horloge haute résolution non ajustable.

Availability : Solaris.

Ajouté dans la version 3.3.

`time.`**`CLOCK_MONOTONIC`**

Horloge qui ne peut pas être réglée et représente l'heure monotone depuis un point de départ non spécifié.

Availability : Unix.

Ajouté dans la version 3.3.

`time.`**`CLOCK_MONOTONIC_RAW`**

Similaire à `CLOCK_MONOTONIC`, mais donne accès à une heure matérielle brute qui n'est pas soumise aux ajustements NTP.

Availability : Linux >= 2.6.28, macOS >= 10.12.

Ajouté dans la version 3.3.

`time.`**`CLOCK_MONOTONIC_RAW_APPROX`**

Similar to `CLOCK_MONOTONIC_RAW`, but reads a value cached by the system at context switch and hence has less accuracy.

Availability : macOS >= 10.12.

Ajouté dans la version 3.13.

`time.`**`CLOCK_PROCESS_CPUTIME_ID`**

Minuterie haute résolution par processus du CPU.

Availability : Unix.

Ajouté dans la version 3.3.

`time.`**`CLOCK_PROF`**

Minuterie haute résolution par processus du CPU.

Availability : FreeBSD, NetBSD >= 7, OpenBSD.

Ajouté dans la version 3.7.

`time.`**`CLOCK_TAI`**

Temps Atomique International (article en anglais)

Le système doit avoir un tableau des secondes intercalaires pour pouvoir donner une réponse correcte. Les logiciels PTP ou NTP savent gérer un tableau des secondes intercalaires.

Availability : Linux.

Ajouté dans la version 3.9.

`time.`**`CLOCK_THREAD_CPUTIME_ID`**

Horloge de temps CPU spécifique au fil d'exécution.

Availability : Unix.

Ajouté dans la version 3.3.

`time.`**`CLOCK_UPTIME`**

Heure dont la valeur absolue correspond à l'heure à laquelle le système a été exécuté et non suspendu, fournissant une mesure précise du temps de disponibilité, à la fois absolue et à intervalle.

Availability : FreeBSD, OpenBSD >= 5.5.

Ajouté dans la version 3.7.

`time.`**`CLOCK_UPTIME_RAW`**

Horloge qui s'incrémente de manière monotone, en suivant le temps écoulé depuis un point arbitraire, n'est pas affectée par les ajustements de fréquence ou de temps et n'est pas incrémentée lorsque le système est en veille.

Availability : macOS >= 10.12.

Ajouté dans la version 3.8.

`time.CLOCK_UPTIME_RAW_APPROX`

Like `CLOCK_UPTIME_RAW`, but the value is cached by the system at context switches and therefore has less accuracy.

Availability : macOS >= 10.12.

Ajouté dans la version 3.13.

La constante suivante est le seul paramètre pouvant être envoyé à `clock_settime()`.

`time.CLOCK_REALTIME`

Horloge en temps réel à l'échelle du système. Le réglage de cette horloge nécessite des privilèges appropriés.

Availability : Unix.

Ajouté dans la version 3.3.

16.3.3 Constantes de fuseau horaire

`time.altzone`

Décalage du fuseau horaire DST local, en secondes à l'ouest de UTC, s'il en est défini un. Cela est négatif si le fuseau horaire DST local est à l'est de UTC (comme en Europe occidentale, y compris le Royaume-Uni). Utilisez ceci uniquement si `daylight` est différent de zéro. Voir note ci-dessous.

`time.daylight`

Non nul si un fuseau horaire DST est défini. Voir note ci-dessous.

`time.timezone`

Décalage du fuseau horaire local (hors heure d'été), en secondes à l'ouest de l'UTC (négatif dans la plupart des pays d'Europe occidentale, positif aux États-Unis, nul au Royaume-Uni). Voir note ci-dessous.

`time.tzname`

Une paire de chaînes : la première est le nom du fuseau horaire local autre que DST, la seconde est le nom du fuseau horaire DST local. Si aucun fuseau horaire DST n'est défini, la deuxième chaîne ne doit pas être utilisée. Voir note ci-dessous.

Note

For the above Timezone constants (`altzone`, `daylight`, `timezone`, and `tzname`), the value is determined by the timezone rules in effect at module load time or the last time `tzset()` is called and may be incorrect for times in the past. It is recommended to use the `tm_gmtoff` and `tm_zone` results from `localtime()` to obtain timezone information.

Voir aussi

Module `datetime`
Interface plus orientée objet vers les dates et les heures.

Module `locale`
Services d'internationalisation. Les paramètres régionaux affectent l'interprétation de nombreux spécificateurs de format dans `strftime()` et `strptime()`.

Module `calendar`
Fonctions générales liées au calendrier. `timegm()` est l'inverse de `gmtime()` à partir de ce module.

Notes

16.4 logging --- Logging facility for Python

Code source : Lib/logging/__init__.py

Important

Cette page contient les informations de référence de l'API. Pour des tutoriels et des discussions sur des sujets plus avancés, voir
- Tutoriel de découverte
- Tutoriel avancé
- Recettes pour la journalisation

Ce module définit les fonctions et les classes qui mettent en œuvre un système flexible d'enregistrement des événements pour les applications et les bibliothèques.

L'intérêt principal d'utiliser une API de journalisation fournie par un module de bibliothèque standard réside dans le fait que tous les modules Python peuvent participer à la journalisation, de sorte que le journal de votre application peut inclure vos propres messages et ceux de modules tiers.

Here's a simple example of idiomatic usage :

```
# myapp.py
import logging
import mylib
logger = logging.getLogger(__name__)

def main():
    logging.basicConfig(filename='myapp.log', level=logging.INFO)
    logger.info('Started')
    mylib.do_something()
    logger.info('Finished')

if __name__ == '__main__':
    main()
```

```
# mylib.py
import logging
logger = logging.getLogger(__name__)

def do_something():
    logger.info('Doing something')
```

Si vous exécutez *myapp.py*, vous verrez ceci dans *myapp.log* :

```
INFO:__main__:Started
INFO:mylib:Doing something
INFO:__main__:Finished
```

The key feature of this idiomatic usage is that the majority of code is simply creating a module level logger with `getLogger(__name__)`, and using that logger to do any needed logging. This is concise, while allowing downstream code fine-grained control if needed. Logged messages to the module-level logger get forwarded to handlers of loggers in higher-level modules, all the way up to the highest-level logger known as the root logger ; this approach is known as hierarchical logging.

For logging to be useful, it needs to be configured : setting the levels and destinations for each logger, potentially changing how specific modules log, often based on command-line arguments or application configuration. In most

cases, like the one above, only the root logger needs to be so configured, since all the lower level loggers at module level eventually forward their messages to its handlers. `basicConfig()` provides a quick way to configure the root logger that handles many use cases.

Ce module offre de nombreuses fonctionnalités et une grande flexibilité. Si vous n'êtes pas familier avec la journalisation, la meilleure façon de l'appréhender est de consulter les tutoriels (**voir les liens ci-dessus et à droite**).

The basic classes defined by the module, together with their attributes and methods, are listed in the sections below.

- les enregistreurs (*loggers* en anglais) exposent l'interface que le code de l'application utilise directement ;
- les gestionnaires (*handlers*) envoient les entrées de journal (créées par les *loggers*) vers les destinations voulues ;
- les filtres (*filters*) fournissent un moyen de choisir plus finement quelles entrées de journal doivent être sorties ;
- les formateurs (*formatters*) définissent la structure de l'entrée de journal dans la sortie finale.

16.4.1 Enregistreurs

Les enregistreurs ont les attributs et les méthodes suivants. Notez que les enregistreurs ne doivent *JAMAIS* être instanciés directement, mais toujours par la fonction au niveau du module `logging.getLogger(nom)`. Les appels à `getLogger()` avec le même nom renvoient toujours une référence au même objet enregistreur.

The `name` is potentially a period-separated hierarchical value, like `foo.bar.baz` (though it could also be just plain `foo`, for example). Loggers that are further down in the hierarchical list are children of loggers higher up in the list. For example, given a logger with a name of `foo`, loggers with names of `foo.bar`, `foo.bar.baz`, and `foo.bam` are all descendants of `foo`. In addition, all loggers are descendants of the root logger. The logger name hierarchy is analogous to the Python package hierarchy, and identical to it if you organise your loggers on a per-module basis using the recommended construction `logging.getLogger(__name__)`. That's because in a module, `__name__` is the module's name in the Python package namespace.

class `logging.`**`Logger`**

`name`

This is the logger's name, and is the value that was passed to `getLogger()` to obtain the logger.

> **Note**
>
> This attribute should be treated as read-only.

`level`

The threshold of this logger, as set by the `setLevel()` method.

> **Note**
>
> Do not set this attribute directly - always use `setLevel()`, which has checks for the level passed to it.

`parent`

The parent logger of this logger. It may change based on later instantiation of loggers which are higher up in the namespace hierarchy.

> **Note**
>
> This value should be treated as read-only.

`propagate`

Si cet attribut est évalué comme vrai, les événements enregistrés dans cet enregistreur seront transmis aux gestionnaires des enregistreurs de niveau supérieur (parents), en plus des gestionnaires attachés à

l'enregistreur. Les messages sont transmis directement aux gestionnaires des enregistreurs parents — ni le niveau ni les filtres des enregistreurs parentaux en question ne sont pris en compte.

S'il s'évalue comme faux, les messages de journalisation ne sont pas transmis aux gestionnaires des enregistreurs parents.

Spelling it out with an example : If the propagate attribute of the logger named `A.B.C` evaluates to true, any event logged to `A.B.C` via a method call such as `logging.getLogger('A.B.C').error(...)` will [subject to passing that logger's level and filter settings] be passed in turn to any handlers attached to loggers named `A.B`, `A` and the root logger, after first being passed to any handlers attached to `A.B.C`. If any logger in the chain `A.B.C`, `A.B`, `A` has its `propagate` attribute set to false, then that is the last logger whose handlers are offered the event to handle, and propagation stops at that point.

Le constructeur fixe cet attribut à `True`.

> **Note**
>
> si vous associez un gestionnaire à un enregistreur *et* à un ou plusieurs de ses parents, il peut émettre le même enregistrement plusieurs fois. En général, il est rare d'avoir besoin d'attacher un gestionnaire à plus d'un enregistreur — si vous l'attachez simplement à l'enregistreur approprié le plus haut possible dans la hiérarchie des enregistreurs, alors il voit tous les événements enregistrés par tous les enregistreurs descendants, à condition que leur paramètre de propagation soit laissé à `True`. La pratique la plus courante est de n'attacher les gestionnaires qu'à l'enregistreur racine et à laisser la propagation s'occuper du reste.

handlers

The list of handlers directly attached to this logger instance.

> **Note**
>
> This attribute should be treated as read-only ; it is normally changed via the *`addHandler()`* and *`removeHandler()`* methods, which use locks to ensure thread-safe operation.

disabled

This attribute disables handling of any events. It is set to `False` in the initializer, and only changed by logging configuration code.

> **Note**
>
> This attribute should be treated as read-only.

setLevel(*level*)

Fixe le seuil de l'enregistreur au niveau *level*. Les messages de journalisation qui sont moins graves que *level* sont ignorés ; les messages qui ont une gravité égale à *level* ou plus élevée sont traités par le ou les gestionnaires de l'enregistreur, à moins que le niveau d'un gestionnaire n'ait été fixé à un niveau de gravité plus élevé que *level*.

Lorsqu'un enregistreur est créé, son niveau est fixé à *`NOTSET`* (ce qui entraîne le traitement de tous les messages lorsque l'enregistreur est l'enregistreur racine, ou la délégation au parent lorsque l'enregistreur est un enregistreur non racine). Notez que l'enregistreur racine est créé avec le niveau *`WARNING`*.

Le terme « délégation au parent » signifie que si un enregistreur a un niveau de `NOTSET`, sa chaîne d'enregistreurs parents est parcourue jusqu'à ce qu'un parent ayant un niveau autre que `NOTSET` soit trouvé, ou que la racine soit atteinte.

Si un parent est trouvé avec un niveau autre que NOTSET, alors le niveau de ce parent est utilisé comme le niveau effectif de l'enregistreur d'où la recherche du parent a commencé. Ce niveau est utilisé pour déterminer comment un événement d'enregistrement est traité.

Si la racine est atteinte, et qu'elle a un niveau de NOTSET, alors tous les messages sont traités. Sinon, le niveau de la racine est utilisé comme niveau effectif.

Voir *Niveaux de journalisation* pour la liste des niveaux.

Modifié dans la version 3.2 : le paramètre *level* peut désormais être une chaîne de caractères représentant le niveau de gravité (comme `'INFO'`) en plus des constantes entières (comme `INFO`). Notez cependant que les niveaux sont stockés en interne sous forme d'entiers, et que des méthodes telles que `getEffectiveLevel()` et `isEnabledFor()` renvoient et s'attendent à recevoir des entiers.

isEnabledFor(*level*)

Indique si un message de gravité *level* est traitable par cet enregistreur. Cette méthode vérifie d'abord le niveau de gravité du module défini par `logging.disable(level)` et ensuite le niveau effectif de l'enregistreur, déterminé par `getEffectiveLevel()`.

getEffectiveLevel()

Indique le niveau effectif de l'enregistreur. Si une valeur autre que `NOTSET` a été définie en utilisant `setLevel()`, elle est renvoyée. Sinon, la hiérarchie est parcourue vers la racine jusqu'à ce qu'une valeur autre que `NOTSET` soit trouvée, et cette valeur est renvoyée. La valeur renvoyée est un entier, par exemple `logging.DEBUG`, `logging.INFO`, etc.

getChild(*suffix*)

Renvoie un enregistreur enfant de l'enregistreur, déterminé par *suffix*. Ainsi, `logging.getLogger('abc').getChild('def.ghi')` renvoie le même enregistreur que `logging.getLogger('abc.def.ghi')`. C'est une méthode destinée à simplifier la vie du développeur. Elle est très utile quand l'enregistreur parent est nommé en utilisant, par exemple, `__name__` plutôt qu'une chaîne de caractères littérale.

Ajouté dans la version 3.2.

getChildren()

Returns a set of loggers which are immediate children of this logger. So for example `logging.getLogger().getChildren()` might return a set containing loggers named `foo` and `bar`, but a logger named `foo.bar` wouldn't be included in the set. Likewise, `logging.getLogger('foo').getChildren()` might return a set including a logger named `foo.bar`, but it wouldn't include one named `foo.bar.baz`.

Ajouté dans la version 3.12.

debug(*msg*, **args*, ***kwargs*)

Enregistre un message de niveau `DEBUG` dans cet enregistreur. *msg* est la chaîne du message qui sera formatée avec *args* en utilisant l'opérateur de formatage. Cela signifie qu'il est possible de mettre des mots-clé dans la chaîne et de passer un dictionnaire en argument. Si *args* n'est pas fourni, aucun formatage « à la % » n'est appliqué.

Quatre mots-clés de *kwargs* sont analysés : *exc_info*, *stack_info*, *stacklevel* et *extra*.

Si la valeur booléenne de *exc_info* est vraie, les informations des exceptions sont ajoutées au message. Si *exc_info* est un *n*-uplet d'exception (au format identique aux valeurs renvoyées par `sys.exc_info()`) ou une instance d'exception, sa valeur est utilisée. Dans le cas contraire, les informations sur l'exception sont déterminées par un appel à `sys.exc_info()`.

Le deuxième argument par mot-clé optionnel est *stack_info*, valant `False` par défaut. S'il est vrai, les informations de la pile d'appels sont ajoutées à l'entrée de journal, en incluant aussi l'appel à la fonction de journalisation. Ce ne sont pas les mêmes informations que celles affichées en définissant *exc_info* : les premières représentent les appels de fonctions successifs, du bas de la pile jusqu'à l'appel de la fonction de journalisation dans le fil d'exécution actuel, alors que les secondes portent des informations sur les appels successifs déclenchés par la levée d'une exception et la recherche de gestionnaires pour cette exception.

Il est possible de définir *stack_info* indépendamment de *exc_info*, p. ex. pour s'assurer que l'exécution a atteint un certain point dans le code, même si aucune exception n'a été levée. La pile d'appels est alors affichée après la ligne d'en-tête suivante :

```
Stack (most recent call last):
```

Elle imite la ligne `Traceback (most recent call last)` : affichée avec la pile d'appels d'une exception.

Le troisième argument par mot-clé optionnel est *stacklevel*, valant `1` par défaut. S'il est supérieur à 1, il correspond au nombre d'entrées dans la pile qui sont ignorées en déterminant le numéro de ligne et le nom de la fonction dans la classe `LogRecord` créée pour l'évènement de journalisation. C'est utile pour les utilitaires de journalisation car cela permet d'ignorer les informations (nom de fonction, fichier source et ligne) de l'utilitaire et de ne traiter que celles de l'appelant. Le nom de ce paramètre est le même que son équivalent dans le module `warnings`.

The fourth keyword argument is *extra* which can be used to pass a dictionary which is used to populate the `__dict__` of the `LogRecord` created for the logging event with user-defined attributes. These custom attributes can then be used as you like. For example, they could be incorporated into logged messages. For example :

```
FORMAT = '%(asctime)s %(clientip)-15s %(user)-8s %(message)s'
logging.basicConfig(format=FORMAT)
d = {'clientip': '192.168.0.1', 'user': 'fbloggs'}
logger = logging.getLogger('tcpserver')
logger.warning('Protocol problem: %s', 'connection reset', extra=d)
```

affiche

```
2006-02-08 22:20:02,165 192.168.0.1 fbloggs  Protocol problem: connection␣
↪reset
```

Les clés du dictionnaire passé dans *extra* ne doivent pas être les mêmes que les clés utilisées par le système de journalisation. Voir la documentation de la classe `Formatter` pour plus de précisions sur les clés utilisées par le système de journalisation.

Si vous choisissez d'utiliser des attributs dans les messages à journaliser, il faut être prudent. Ainsi, dans l'exemple précédent, le `Formatter` a été configuré avec une chaîne qui attend *clientip* et *user* dans le dictionnaire d'attributs du `LogRecord`. S'ils sont manquants, le message n'est pas enregistré car une exception de formatage de chaîne est levée. Il faut alors toujours passer un dictionnaire *extra* avec ces clés.

Même si elle peut sembler gênante, cette fonctionnalité est nécessaire dans certains cas, comme sur des serveurs à fils d'exécution multiples, où le même code s'exécute dans des contextes différents et où les évènements significatifs dépendent du contexte (comme l'adresse IP du client et le nom d'utilisateur dans l'exemple précédent). Dans ces circonstances, il est clair que les classes `Formatter`s spécialisées doivent être utilisées avec des `Handler`s particuliers.

If no handler is attached to this logger (or any of its ancestors, taking into account the relevant `Logger.propagate` attributes), the message will be sent to the handler set on `lastResort`.

Modifié dans la version 3.2 : ajout du paramètre *stack_info*.

Modifié dans la version 3.5 : le paramètre *exc_info* peut être une instance d'exception.

Modifié dans la version 3.8 : ajout du paramètre *stacklevel*.

info(*msg*, **args*, ***kwargs*)

Enregistre un message avec le niveau de gravité `INFO`. Les arguments ont la même signification que pour `debug()`.

warning(*msg*, **args*, ***kwargs*)

Enregistre un message avec le niveau de gravité `WARNING`. Les arguments ont la même signification que pour `debug()`.

> **Note**
>
> Il existe une méthode obsolète `warn` qui est identique à `warning`. `warn` n'est plus maintenue, prière de ne plus l'utiliser et de la remplacer par `warning`.

error(*msg*, **args*, ***kwargs*)

Enregistre un message avec le niveau de gravité `ERROR`. Les arguments ont la même signification que pour `debug()`.

critical(*msg*, **args*, ***kwargs*)

Enregistre un message avec le niveau de gravité `CRITICAL`. Les arguments ont la même signification que pour `debug()`.

log(*level*, *msg*, **args*, ***kwargs*)

Enregistre un message avec le niveau de gravité *level*. Les arguments ont la même signification que pour `debug()`.

exception(*msg*, **args*, ***kwargs*)

Enregistre un message avec le niveau de gravité `ERROR`. Les arguments ont la même signification que pour `debug()`. Des informations sur l'exception sont ajoutées au message. Cette méthode doit être appelée depuis un gestionnaire d'exceptions.

addFilter(*filter*)

Ajoute le filtre *filter* à l'enregistreur.

removeFilter(*filter*)

Retire le filtre *filter* de cet enregistreur.

filter(*record*)

Applique les filtres associés à l'enregistreur et renvoie `True` si l'entrée doit être traitée. Les filtres sont appliqués les uns après les autres, jusqu'à ce que l'un renvoie faux. Si ce n'est pas le cas, le traitement de l'entrée se poursuit (elle est alors passée aux gestionnaires). Si l'un d'entre eux renvoie faux, le traitement de l'entrée s'arrête.

addHandler(*hdlr*)

Ajoute le gestionnaire *hdlr* à l'enregistreur.

removeHandler(*hdlr*)

Retire le gestionnaire *hdlr* de l'enregistreur.

findCaller(*stack_info=False*, *stacklevel=1*)

Détermine le fichier source et la ligne de l'appelant. Renvoie un quadruplet contenant le nom du fichier source, le numéro de ligne, le nom de la fonction et la pile d'appels. La pile vaut `None` si *stack_info* n'est pas `True`.

Le paramètre *stacklevel* est passé par le code appelant `debug()` (entre autres). S'il est supérieur à 1, *n*-1 entrées de la pile sont supprimées avant de renvoyer la pile. C'est pratique quand on appelle des APIs de journalisation à travers du code d'encapsulation car cela permet de retirer les informations sur ce code de l'entrée, tout en conservant celles sur le code au-dessus du code d'encapsulation.

handle(*record*)

Traite une entrée en la passant à tous les gestionnaires de l'enregistreur et à tous ses parents (jusqu'à ce qu'un *propagate* soit faux). C'est pratique pour désérialiser les entrées reçues d'un connecteur et celles créées localement. Du filtrage au niveau de l'enregistreur est appliqué en appelant `filter()`.

makeRecord(*name*, *level*, *fn*, *lno*, *msg*, *args*, *exc_info*, *func=None*, *extra=None*, *sinfo=None*)

Fabrique qui peut être redéfinie pour créer des instances de `LogRecord`.

hasHandlers()

Vérifie si l'enregistreur a des gestionnaires associés. Elle recherche les gestionnaires de l'enregistreur et ceux de ses parents. Renvoie `True` si au moins un gestionnaire a été trouvé et `False` sinon. Cette méthode

arrête de remonter la hiérarchie dès qu'un enregistreur avec l'attribut *propagate* à faux est rencontré — cet enregistreur est alors le dernier dans lequel la méthode cherche des gestionnaires.

Ajouté dans la version 3.2.

Modifié dans la version 3.7 : les enregistreurs peuvent être sérialisés et désérialisés.

16.4.2 Niveaux de journalisation

Les valeurs numériques des niveaux de journalisation sont données dans le tableau suivant. Celles-ci n'ont d'intérêt que si vous voulez définir vos propres niveaux, avec des valeurs spécifiques par rapport aux niveaux prédéfinis. Si vous définissez un niveau avec la même valeur numérique, il écrase la valeur prédéfinie ; le nom prédéfini est perdu.

Niveau	Valeur numérique	What it means / When to use it
`logging.NOTSET`	0	When set on a logger, indicates that ancestor loggers are to be consulted to determine the effective level. If that still resolves to `NOTSET`, then all events are logged. When set on a handler, all events are handled.
`logging.DEBUG`	10	Detailed information, typically only of interest to a developer trying to diagnose a problem.
`logging.INFO`	20	Confirmation que tout fonctionne comme prévu.
`logging.WARNING`	30	An indication that something unexpected happened, or that a problem might occur in the near future (e.g. 'disk space low'). The software is still working as expected.
`logging.ERROR`	40	Du fait d'un problème plus sérieux, le logiciel n'a pas été capable de réaliser une tâche.
`logging.CRITICAL`	50	Une erreur sérieuse, indiquant que le programme lui-même pourrait être incapable de continuer à fonctionner.

16.4.3 Gestionnaires

Handlers have the following attributes and methods. Note that `Handler` is never instantiated directly ; this class acts as a base for more useful subclasses. However, the `__init__()` method in subclasses needs to call `Handler.__init__()`.

class logging.**Handler**

__init__(*level=NOTSET*)

Initialise l'objet `Handler` en définissant le niveau de gravité, en initialisant la liste des filtres avec une liste vide et en créant un verrou (avec `createLock()`) pour sérialiser l'accès au mécanisme d'E-S.

createLock()

Instancie un verrou qui peut être utilisé pour sérialiser l'accès au système d'E-S sous-jacent (qui peut ne pas être à fil d'exécution sécurisé).

acquire()

Acquiert le verrou créé par `createLock()`.

release()

Relâche le verrou acquis par `acquire()`.

setLevel(*level*)

Ajuste le seuil déclenchement du gestionnaire au niveau *level*. Les messages de gravité moindre que *level* sont alors ignorés. Ce seuil est fixé à `NOTSET` lors de la création d'un gestionnaire (ce qui signifie que tous les messages seront traités).

Voir *Niveaux de journalisation* pour la liste des niveaux.

Modifié dans la version 3.2 : le paramètre *level* peut être une chaîne de caractères, comme `'INFO'`, en plus d'une constante entière comme `INFO`.

setFormatter(*fmt*)

Définit le `Formatter` du gestionnaire à *fmt*.

addFilter(*filter*)

Ajoute le filtre *filter* au gestionnaire.

removeFilter(*filter*)

Retire le filtre *filter* du gestionnaire.

filter(*record*)

Applique les filtres du gestionnaire à *record* et renvoie `True` si l'entrée doit être traitée. Les filtres sont appliqués l'un après l'autre, jusqu'à ce que l'un renvoie faux. Si aucun d'entre eux ne renvoie faux, l'entrée est enregistrée, sinon le gestionnaire ne traitera pas l'entrée.

flush()

Oblige toutes les entrées à être traitées. Cette fonction ne fait rien de spécial et doit être redéfinie par les sous-classes.

close()

Tidy up any resources used by the handler. This version does no output but removes the handler from an internal map of handlers, which is used for handler lookup by name.

Subclasses should ensure that this gets called from overridden `close()` methods.

handle(*record*)

Traite ou non *record* selon les filtres ajoutés au gestionnaire. Un verrou sur l'E-S. est mis en place durant l'écriture effective.

handleError(*record*)

This method should be called from handlers when an exception is encountered during an `emit()` call. If the module-level attribute `raiseExceptions` is `False`, exceptions get silently ignored. This is what is mostly wanted for a logging system - most users will not care about errors in the logging system, they are more interested in application errors. You could, however, replace this with a custom handler if you wish. The specified record is the one which was being processed when the exception occurred. (The default value of `raiseExceptions` is `True`, as that is more useful during development).

format(*record*)

Formate *record* avec le formateur défini. S'il n'y en a pas, le formateur par défaut du module est utilisé.

emit(*record*)

Journalise *record* « pour de bon ». Cette version doit être redéfinie par les sous-classes et lève donc une `NotImplementedError`.

> **Avertissement**
>
> This method is called after a handler-level lock is acquired, which is released after this method returns. When you override this method, note that you should be careful when calling anything that invokes other parts of the logging API which might do locking, because that might result in a deadlock. Specifically :

— Logging configuration APIs acquire the module-level lock, and then individual handler-level locks as those handlers are configured.
— Many logging APIs lock the module-level lock. If such an API is called from this method, it could cause a deadlock if a configuration call is made on another thread, because that thread will try to acquire the module-level lock *before* the handler-level lock, whereas this thread tries to acquire the module-level lock *after* the handler-level lock (because in this method, the handler-level lock has already been acquired).

Les gestionnaires de la bibliothèque standard sont répertoriés dans `logging.handlers`.

16.4.4 Formateurs

class `logging.`**Formatter**(*fmt=None*, *datefmt=None*, *style='%'*, *validate=True*, **, defaults=None*)

Responsible for converting a `LogRecord` to an output string to be interpreted by a human or external system.

Paramètres

— **fmt** (`str`) -- A format string in the given *style* for the logged output as a whole. The possible mapping keys are drawn from the `LogRecord` object's *LogRecord attributes*. If not specified, `'%(message)s'` is used, which is just the logged message.
— **datefmt** (`str`) -- A format string in the given *style* for the date/time portion of the logged output. If not specified, the default described in `formatTime()` is used.
— **style** (`str`) -- Can be one of `'%'`, `'{'` or `'$'` and determines how the format string will be merged with its data : using one of *Formatage de chaines à la printf* (`%`), `str.format()` (`{`) or `string.Template` (`$`). This only applies to *fmt* and *datefmt* (e.g. `'%(message)s'` versus `'{message}'`), not to the actual log messages passed to the logging methods. However, there are other ways to use `{`- and `$`-formatting for log messages.
— **validate** (`bool`) -- If `True` (the default), incorrect or mismatched *fmt* and *style* will raise a `ValueError`; for example, `logging.Formatter('%(asctime)s - %(message)s', style='{')`.
— **defaults** (`dict[str, Any]`) -- A dictionary with default values to use in custom fields. For example, `logging.Formatter('%(ip)s %(message)s', defaults={"ip": None})`

Modifié dans la version 3.2 : Added the *style* parameter.

Modifié dans la version 3.8 : Added the *validate* parameter.

Modifié dans la version 3.10 : Added the *defaults* parameter.

format(*record*)

The record's attribute dictionary is used as the operand to a string formatting operation. Returns the resulting string. Before formatting the dictionary, a couple of preparatory steps are carried out. The *message* attribute of the record is computed using *msg % args*. If the formatting string contains `'(asctime)'`, `formatTime()` is called to format the event time. If there is exception information, it is formatted using `formatException()` and appended to the message. Note that the formatted exception information is cached in attribute *exc_text*. This is useful because the exception information can be pickled and sent across the wire, but you should be careful if you have more than one `Formatter` subclass which customizes the formatting of exception information. In this case, you will have to clear the cached value (by setting the *exc_text* attribute to `None`) after a formatter has done its formatting, so that the next formatter to handle the event doesn't use the cached value, but recalculates it afresh.

If stack information is available, it's appended after the exception information, using `formatStack()` to transform it if necessary.

formatTime(*record*, *datefmt=None*)

This method should be called from `format()` by a formatter which wants to make use of a formatted time. This method can be overridden in formatters to provide for any specific requirement, but the basic behavior is as follows : if *datefmt* (a string) is specified, it is used with `time.strftime()` to format the creation time of the record. Otherwise, the format '%Y-%m-%d %H :%M :%S,uuu' is used, where the uuu part is a millisecond value and the other letters are as per the `time.strftime()` documentation. An example time in this format is `2003-01-23 00:29:50,411`. The resulting string is returned.

This function uses a user-configurable function to convert the creation time to a tuple. By default, `time.localtime()` is used ; to change this for a particular formatter instance, set the `converter` attribute to a function with the same signature as `time.localtime()` or `time.gmtime()`. To change it for all formatters, for example if you want all logging times to be shown in GMT, set the `converter` attribute in the `Formatter` class.

Modifié dans la version 3.3 : Previously, the default format was hard-coded as in this example : `2010-09-06 22:38:15,292` where the part before the comma is handled by a strptime format string (`'%Y-%m-%d %H:%M:%S'`), and the part after the comma is a millisecond value. Because strptime does not have a format placeholder for milliseconds, the millisecond value is appended using another format string, `'%s,%03d'` --- and both of these format strings have been hardcoded into this method. With the change, these strings are defined as class-level attributes which can be overridden at the instance level when desired. The names of the attributes are `default_time_format` (for the strptime format string) and `default_msec_format` (for appending the millisecond value).

Modifié dans la version 3.9 : The `default_msec_format` can be `None`.

formatException(*exc_info*)

Formats the specified exception information (a standard exception tuple as returned by `sys.exc_info()`) as a string. This default implementation just uses `traceback.print_exception()`. The resulting string is returned.

formatStack(*stack_info*)

Formats the specified stack information (a string as returned by `traceback.print_stack()`, but with the last newline removed) as a string. This default implementation just returns the input value.

class logging.**BufferingFormatter**(*linefmt=None*)

A base formatter class suitable for subclassing when you want to format a number of records. You can pass a `Formatter` instance which you want to use to format each line (that corresponds to a single record). If not specified, the default formatter (which just outputs the event message) is used as the line formatter.

formatHeader(*records*)

Return a header for a list of *records*. The base implementation just returns the empty string. You will need to override this method if you want specific behaviour, e.g. to show the count of records, a title or a separator line.

formatFooter(*records*)

Return a footer for a list of *records*. The base implementation just returns the empty string. You will need to override this method if you want specific behaviour, e.g. to show the count of records or a separator line.

format(*records*)

Return formatted text for a list of *records*. The base implementation just returns the empty string if there are no records ; otherwise, it returns the concatenation of the header, each record formatted with the line formatter, and the footer.

16.4.5 Filtres

`Filters` can be used by `Handlers` and `Loggers` for more sophisticated filtering than is provided by levels. The base filter class only allows events which are below a certain point in the logger hierarchy. For example, a filter initialized with 'A.B' will allow events logged by loggers 'A.B', 'A.B.C', 'A.B.C.D', 'A.B.D' etc. but not 'A.BB', 'B.A.B' etc. If initialized with the empty string, all events are passed.

class logging.**Filter**(*name=''*)

Returns an instance of the `Filter` class. If *name* is specified, it names a logger which, together with its children, will have its events allowed through the filter. If *name* is the empty string, allows every event.

filter(*record*)

Is the specified record to be logged ? Returns false for no, true for yes. Filters can either modify log records in-place or return a completely different record instance which will replace the original log record in any future processing of the event.

Note that filters attached to handlers are consulted before an event is emitted by the handler, whereas filters attached to loggers are consulted whenever an event is logged (using `debug()`, `info()`, etc.), before sending an event to handlers. This means that events which have been generated by descendant loggers will not be filtered by a logger's filter setting, unless the filter has also been applied to those descendant loggers.

You don't actually need to subclass `Filter` : you can pass any instance which has a `filter` method with the same semantics.

Modifié dans la version 3.2 : You don't need to create specialized `Filter` classes, or use other classes with a `filter` method : you can use a function (or other callable) as a filter. The filtering logic will check to see if the filter object has a `filter` attribute : if it does, it's assumed to be a `Filter` and its `filter()` method is called. Otherwise, it's assumed to be a callable and called with the record as the single parameter. The returned value should conform to that returned by `filter()`.

Modifié dans la version 3.12 : You can now return a `LogRecord` instance from filters to replace the log record rather than modifying it in place. This allows filters attached to a `Handler` to modify the log record before it is emitted, without having side effects on other handlers.

Although filters are used primarily to filter records based on more sophisticated criteria than levels, they get to see every record which is processed by the handler or logger they're attached to : this can be useful if you want to do things like counting how many records were processed by a particular logger or handler, or adding, changing or removing attributes in the `LogRecord` being processed. Obviously changing the LogRecord needs to be done with some care, but it does allow the injection of contextual information into logs (see filters-contextual).

16.4.6 Objets LogRecord

`LogRecord` instances are created automatically by the `Logger` every time something is logged, and can be created manually via `makeLogRecord()` (for example, from a pickled event received over the wire).

class `logging.`**`LogRecord`**(*name*, *level*, *pathname*, *lineno*, *msg*, *args*, *exc_info*, *func=None*, *sinfo=None*)

Contains all the information pertinent to the event being logged.

The primary information is passed in *msg* and *args*, which are combined using `msg % args` to create the `message` attribute of the record.

Paramètres

- **name** (`str`) -- The name of the logger used to log the event represented by this `LogRecord`. Note that the logger name in the `LogRecord` will always have this value, even though it may be emitted by a handler attached to a different (ancestor) logger.
- **level** (`int`) -- The *numeric level* of the logging event (such as `10` for `DEBUG`, `20` for `INFO`, etc). Note that this is converted to *two* attributes of the LogRecord : `levelno` for the numeric value and `levelname` for the corresponding level name.
- **pathname** (`str`) -- The full string path of the source file where the logging call was made.
- **lineno** (`int`) -- The line number in the source file where the logging call was made.
- **msg** (`Any`) -- The event description message, which can be a %-format string with place-holders for variable data, or an arbitrary object (see arbitrary-object-messages).
- **args** (`tuple | dict[str, Any]`) -- Variable data to merge into the *msg* argument to obtain the event description.
- **exc_info** (`tuple[type[BaseException], BaseException, types.TracebackType] | None`) -- An exception tuple with the current exception information, as returned by `sys.exc_info()`, or `None` if no exception information is available.
- **func** (`str | None`) -- The name of the function or method from which the logging call was invoked.
- **sinfo** (`str | None`) -- A text string representing stack information from the base of the stack in the current thread, up to the logging call.

getMessage()

Returns the message for this `LogRecord` instance after merging any user-supplied arguments with the message. If the user-supplied message argument to the logging call is not a string, `str()` is called on it to convert it to a string. This allows use of user-defined classes as messages, whose `__str__` method can return the actual format string to be used.

Modifié dans la version 3.2 : The creation of a *LogRecord* has been made more configurable by providing a factory which is used to create the record. The factory can be set using *getLogRecordFactory()* and *setLogRecordFactory()* (see this for the factory's signature).

This functionality can be used to inject your own values into a *LogRecord* at creation time. You can use the following pattern :

```
old_factory = logging.getLogRecordFactory()

def record_factory(*args, **kwargs):
    record = old_factory(*args, **kwargs)
    record.custom_attribute = 0xdecafbad
    return record

logging.setLogRecordFactory(record_factory)
```

With this pattern, multiple factories could be chained, and as long as they don't overwrite each other's attributes or unintentionally overwrite the standard attributes listed above, there should be no surprises.

16.4.7 LogRecord attributes

The LogRecord has a number of attributes, most of which are derived from the parameters to the constructor. (Note that the names do not always correspond exactly between the LogRecord constructor parameters and the LogRecord attributes.) These attributes can be used to merge data from the record into the format string. The following table lists (in alphabetical order) the attribute names, their meanings and the corresponding placeholder in a %-style format string.

If you are using {}-formatting (*str.format()*), you can use `{attrname}` as the placeholder in the format string. If you are using $-formatting (*string.Template*), use the form `${attrname}`. In both cases, of course, replace `attrname` with the actual attribute name you want to use.

In the case of {}-formatting, you can specify formatting flags by placing them after the attribute name, separated from it with a colon. For example : a placeholder of `{msecs:03.0f}` would format a millisecond value of `4` as `004`. Refer to the *str.format()* documentation for full details on the options available to you.

Attribute name	Format	Description
args	You shouldn't need to format this yourself.	The tuple of arguments merged into `msg` to produce `message`, or a dict whose values are used for the merge (when there is only one argument, and it is a dictionary).
asctime	`%(asctime)s`	Human-readable time when the *LogRecord* was created. By default this is of the form '2003-07-08 16 :49 :45,896' (the numbers after the comma are millisecond portion of the time).
created	`%(created)f`	Time when the *LogRecord* was created (as returned by *time.time_ns()* / 1e9).
exc_info	You shouldn't need to format this yourself.	Exception tuple (à la `sys.exc_info`) or, if no exception has occurred, `None`.
filename	`%(filename)s`	Filename portion of `pathname`.
funcName	`%(funcName)s`	Name of function containing the logging call.
levelname	`%(levelname)s`	Text logging level for the message (`'DEBUG'`, `'INFO'`, `'WARNING'`, `'ERROR'`, `'CRITICAL'`).
levelno	`%(levelno)s`	Numeric logging level for the message (*DEBUG*, *INFO*, *WARNING*, *ERROR*, *CRITICAL*).
lineno	`%(lineno)d`	Source line number where the logging call was issued (if available).
message	`%(message)s`	The logged message, computed as `msg % args`. This is set when *Formatter.format()* is invoked.
module	`%(module)s`	Module (name portion of `filename`).
msecs	`%(msecs)d`	Millisecond portion of the time when the *LogRecord* was created.
msg	You shouldn't need to format this yourself.	The format string passed in the original logging call. Merged with `args` to produce `message`, or an arbitrary object (see arbitrary-object-messages).
name	`%(name)s`	Name of the logger used to log the call.
pathname	`%(pathname)s`	Full pathname of the source file where the logging call was issued (if available).
process	`%(process)d`	Process ID (if available).
processName	`%(processName)`	Process name (if available).
relativeCreated	`%(relativeCrea`	Time in milliseconds when the LogRecord was created, relative to the time the logging module was loaded.
stack_info	You shouldn't need to format this yourself.	Stack frame information (where available) from the bottom of the stack in the current thread, up to and including the stack frame of the logging call which resulted in the creation of this record.
thread	`%(thread)d`	Thread ID (if available).
threadName	`%(threadName)s`	Thread name (if available).
taskName	`%(taskName)s`	*asyncio.Task* name (if available).

Modifié dans la version 3.1 : *processName* was added.

Modifié dans la version 3.12 : *taskName* was added.

16.4.8 LoggerAdapter Objects

`LoggerAdapter` instances are used to conveniently pass contextual information into logging calls. For a usage example, see the section on adding contextual information to your logging output.

class `logging.`**`LoggerAdapter`**(*logger*, *extra*, *merge_extra=False*)

Returns an instance of `LoggerAdapter` initialized with an underlying `Logger` instance, a dict-like object (*extra*), and a boolean (*merge_extra*) indicating whether or not the *extra* argument of individual log calls should be merged with the `LoggerAdapter` extra. The default behavior is to ignore the *extra* argument of individual log calls and only use the one of the `LoggerAdapter` instance

process(*msg*, *kwargs*)

Modifies the message and/or keyword arguments passed to a logging call in order to insert contextual information. This implementation takes the object passed as *extra* to the constructor and adds it to *kwargs* using key 'extra'. The return value is a (*msg*, *kwargs*) tuple which has the (possibly modified) versions of the arguments passed in.

manager

Delegates to the underlying `manager` on *logger*.

_log

Delegates to the underlying `_log()` method on *logger*.

In addition to the above, `LoggerAdapter` supports the following methods of `Logger` : `debug()`, `info()`, `warning()`, `error()`, `exception()`, `critical()`, `log()`, `isEnabledFor()`, `getEffectiveLevel()`, `setLevel()` and `hasHandlers()`. These methods have the same signatures as their counterparts in `Logger`, so you can use the two types of instances interchangeably.

Modifié dans la version 3.2 : The `isEnabledFor()`, `getEffectiveLevel()`, `setLevel()` and `hasHandlers()` methods were added to `LoggerAdapter`. These methods delegate to the underlying logger.

Modifié dans la version 3.6 : Attribute `manager` and method `_log()` were added, which delegate to the underlying logger and allow adapters to be nested.

Modifié dans la version 3.13 : The *merge_extra* argument was added.

16.4.9 Thread Safety

The logging module is intended to be thread-safe without any special work needing to be done by its clients. It achieves this though using threading locks ; there is one lock to serialize access to the module's shared data, and each handler also creates a lock to serialize access to its underlying I/O.

If you are implementing asynchronous signal handlers using the `signal` module, you may not be able to use logging from within such handlers. This is because lock implementations in the `threading` module are not always re-entrant, and so cannot be invoked from such signal handlers.

16.4.10 Fonctions de niveau module

In addition to the classes described above, there are a number of module-level functions.

`logging.`**`getLogger`**(*name=None*)

Return a logger with the specified name or, if name is `None`, return the root logger of the hierarchy. If specified, the name is typically a dot-separated hierarchical name like *'a'*, *'a.b'* or *'a.b.c.d'*. Choice of these names is entirely up to the developer who is using logging, though it is recommended that `__name__` be used unless you have a specific reason for not doing that, as mentioned in *Enregistreurs*.

All calls to this function with a given name return the same logger instance. This means that logger instances never need to be passed between different parts of an application.

`logging.`**`getLoggerClass`**()

Return either the standard `Logger` class, or the last class passed to `setLoggerClass()`. This function may be called from within a new class definition, to ensure that installing a customized `Logger` class will not undo customizations already applied by other code. For example :

```
class MyLogger(logging.getLoggerClass()):
    # ... override behaviour here
```

logging.**getLogRecordFactory**()

Return a callable which is used to create a `LogRecord`.

Ajouté dans la version 3.2 : This function has been provided, along with `setLogRecordFactory()`, to allow developers more control over how the `LogRecord` representing a logging event is constructed.

See `setLogRecordFactory()` for more information about the how the factory is called.

logging.**debug**(*msg*, **args*, ***kwargs*)

This is a convenience function that calls `Logger.debug()`, on the root logger. The handling of the arguments is in every way identical to what is described in that method.

The only difference is that if the root logger has no handlers, then `basicConfig()` is called, prior to calling `debug` on the root logger.

For very short scripts or quick demonstrations of `logging` facilities, `debug` and the other module-level functions may be convenient. However, most programs will want to carefully and explicitly control the logging configuration, and should therefore prefer creating a module-level logger and calling `Logger.debug()` (or other level-specific methods) on it, as described at the beginning of this documentation.

logging.**info**(*msg*, **args*, ***kwargs*)

Logs a message with level `INFO` on the root logger. The arguments and behavior are otherwise the same as for `debug()`.

logging.**warning**(*msg*, **args*, ***kwargs*)

Logs a message with level `WARNING` on the root logger. The arguments and behavior are otherwise the same as for `debug()`.

> **Note**
>
> There is an obsolete function `warn` which is functionally identical to `warning`. As `warn` is deprecated, please do not use it - use `warning` instead.

logging.**error**(*msg*, **args*, ***kwargs*)

Logs a message with level `ERROR` on the root logger. The arguments and behavior are otherwise the same as for `debug()`.

logging.**critical**(*msg*, **args*, ***kwargs*)

Logs a message with level `CRITICAL` on the root logger. The arguments and behavior are otherwise the same as for `debug()`.

logging.**exception**(*msg*, **args*, ***kwargs*)

Logs a message with level `ERROR` on the root logger. The arguments and behavior are otherwise the same as for `debug()`. Exception info is added to the logging message. This function should only be called from an exception handler.

logging.**log**(*level*, *msg*, **args*, ***kwargs*)

Logs a message with level *level* on the root logger. The arguments and behavior are otherwise the same as for `debug()`.

logging.**disable**(*level=CRITICAL*)

Provides an overriding level *level* for all loggers which takes precedence over the logger's own level. When the need arises to temporarily throttle logging output down across the whole application, this function can be useful. Its effect is to disable all logging calls of severity *level* and below, so that if you call it with a value of INFO, then all INFO and DEBUG events would be discarded, whereas those of severity WARNING and above would be processed according to the logger's effective level. If `logging.disable(logging.NOTSET)` is

called, it effectively removes this overriding level, so that logging output again depends on the effective levels of individual loggers.

Note that if you have defined any custom logging level higher than `CRITICAL` (this is not recommended), you won't be able to rely on the default value for the *level* parameter, but will have to explicitly supply a suitable value.

Modifié dans la version 3.7 : The *level* parameter was defaulted to level `CRITICAL`. See bpo-28524 for more information about this change.

`logging.addLevelName(level, levelName)`

Associates level *level* with text *levelName* in an internal dictionary, which is used to map numeric levels to a textual representation, for example when a `Formatter` formats a message. This function can also be used to define your own levels. The only constraints are that all levels used must be registered using this function, levels should be positive integers and they should increase in increasing order of severity.

> **Note**
>
> If you are thinking of defining your own levels, please see the section on custom-levels.

`logging.getLevelNamesMapping()`

Returns a mapping from level names to their corresponding logging levels. For example, the string "CRITICAL" maps to `CRITICAL`. The returned mapping is copied from an internal mapping on each call to this function.

Ajouté dans la version 3.11.

`logging.getLevelName(level)`

Returns the textual or numeric representation of logging level *level*.

If *level* is one of the predefined levels `CRITICAL`, `ERROR`, `WARNING`, `INFO` or `DEBUG` then you get the corresponding string. If you have associated levels with names using `addLevelName()` then the name you have associated with *level* is returned. If a numeric value corresponding to one of the defined levels is passed in, the corresponding string representation is returned.

The *level* parameter also accepts a string representation of the level such as 'INFO'. In such cases, this functions returns the corresponding numeric value of the level.

If no matching numeric or string value is passed in, the string 'Level %s' % level is returned.

> **Note**
>
> Levels are internally integers (as they need to be compared in the logging logic). This function is used to convert between an integer level and the level name displayed in the formatted log output by means of the `%(levelname)s` format specifier (see *LogRecord attributes*), and vice versa.

Modifié dans la version 3.4 : In Python versions earlier than 3.4, this function could also be passed a text level, and would return the corresponding numeric value of the level. This undocumented behaviour was considered a mistake, and was removed in Python 3.4, but reinstated in 3.4.2 due to retain backward compatibility.

`logging.getHandlerByName(name)`

Returns a handler with the specified *name*, or `None` if there is no handler with that name.

Ajouté dans la version 3.12.

`logging.getHandlerNames()`

Returns an immutable set of all known handler names.

Ajouté dans la version 3.12.

`logging.makeLogRecord(attrdict)`

Creates and returns a new `LogRecord` instance whose attributes are defined by *attrdict*. This function is useful

for taking a pickled `LogRecord` attribute dictionary, sent over a socket, and reconstituting it as a `LogRecord` instance at the receiving end.

`logging.basicConfig(**kwargs)`

Does basic configuration for the logging system by creating a `StreamHandler` with a default `Formatter` and adding it to the root logger. The functions `debug()`, `info()`, `warning()`, `error()` and `critical()` will call `basicConfig()` automatically if no handlers are defined for the root logger.

This function does nothing if the root logger already has handlers configured, unless the keyword argument *force* is set to `True`.

> **Note**
>
> This function should be called from the main thread before other threads are started. In versions of Python prior to 2.7.1 and 3.2, if this function is called from multiple threads, it is possible (in rare circumstances) that a handler will be added to the root logger more than once, leading to unexpected results such as messages being duplicated in the log.

The following keyword arguments are supported.

Format	Description
filename	Specifies that a `FileHandler` be created, using the specified filename, rather than a `StreamHandler`.
filemode	If *filename* is specified, open the file in this *mode*. Defaults to `'a'`.
format	Use the specified format string for the handler. Defaults to attributes `levelname`, `name` and `message` separated by colons.
datefmt	Use the specified date/time format, as accepted by `time.strftime()`.
style	If *format* is specified, use this style for the format string. One of `'%'`, `'{'` or `'$'` for *printf-style*, `str.format()` or `string.Template` respectively. Defaults to `'%'`.
level	Set the root logger level to the specified *level*.
stream	Use the specified stream to initialize the `StreamHandler`. Note that this argument is incompatible with *filename* - if both are present, a `ValueError` is raised.
handlers	If specified, this should be an iterable of already created handlers to add to the root logger. Any handlers which don't already have a formatter set will be assigned the default formatter created in this function. Note that this argument is incompatible with *filename* or *stream* - if both are present, a `ValueError` is raised.
force	If this keyword argument is specified as true, any existing handlers attached to the root logger are removed and closed, before carrying out the configuration as specified by the other arguments.
encoding	If this keyword argument is specified along with *filename*, its value is used when the `FileHandler` is created, and thus used when opening the output file.
errors	If this keyword argument is specified along with *filename*, its value is used when the `FileHandler` is created, and thus used when opening the output file. If not specified, the value 'backslashreplace' is used. Note that if `None` is specified, it will be passed as such to `open()`, which means that it will be treated the same as passing 'errors'.

Modifié dans la version 3.2 : The *style* argument was added.

Modifié dans la version 3.3 : The *handlers* argument was added. Additional checks were added to catch situations where incompatible arguments are specified (e.g. *handlers* together with *stream* or *filename*, or *stream* together with *filename*).

Modifié dans la version 3.8 : The *force* argument was added.

Modifié dans la version 3.9 : The *encoding* and *errors* arguments were added.

`logging.shutdown()`

Informs the logging system to perform an orderly shutdown by flushing and closing all handlers. This should

be called at application exit and no further use of the logging system should be made after this call.

When the logging module is imported, it registers this function as an exit handler (see `atexit`), so normally there's no need to do that manually.

`logging.setLoggerClass(klass)`

Tells the logging system to use the class *klass* when instantiating a logger. The class should define `__init__()` such that only a name argument is required, and the `__init__()` should call `Logger.__init__()`. This function is typically called before any loggers are instantiated by applications which need to use custom logger behavior. After this call, as at any other time, do not instantiate loggers directly using the subclass : continue to use the `logging.getLogger()` API to get your loggers.

`logging.setLogRecordFactory(factory)`

Set a callable which is used to create a `LogRecord`.

Paramètres

factory -- The factory callable to be used to instantiate a log record.

Ajouté dans la version 3.2 : This function has been provided, along with `getLogRecordFactory()`, to allow developers more control over how the `LogRecord` representing a logging event is constructed.

The factory has the following signature :

```
factory(name, level, fn, lno, msg, args, exc_info, func=None, sinfo=None,
**kwargs)
```

nom
The logger name.

level
The logging level (numeric).

fn
The full pathname of the file where the logging call was made.

lno
The line number in the file where the logging call was made.

msg
The logging message.

args
The arguments for the logging message.

exc_info
An exception tuple, or `None`.

func
The name of the function or method which invoked the logging call.

sinfo
A stack traceback such as is provided by `traceback.print_stack()`, showing the call hierarchy.

kwargs
Additional keyword arguments.

16.4.11 Module-Level Attributes

`logging.lastResort`

A "handler of last resort" is available through this attribute. This is a `StreamHandler` writing to `sys.stderr` with a level of `WARNING`, and is used to handle logging events in the absence of any logging configuration. The end result is to just print the message to `sys.stderr`. This replaces the earlier error message saying that "no handlers could be found for logger XYZ". If you need the earlier behaviour for some reason, `lastResort` can be set to `None`.

Ajouté dans la version 3.2.

`logging.raiseExceptions`

Used to see if exceptions during handling should be propagated.

Default : `True`.

If `raiseExceptions` is `False`, exceptions get silently ignored. This is what is mostly wanted for a logging system - most users will not care about errors in the logging system, they are more interested in application errors.

16.4.12 Integration with the warnings module

The `captureWarnings()` function can be used to integrate `logging` with the `warnings` module.

`logging.captureWarnings`(*capture*)

This function is used to turn the capture of warnings by logging on and off.

If *capture* is `True`, warnings issued by the `warnings` module will be redirected to the logging system. Specifically, a warning will be formatted using `warnings.formatwarning()` and the resulting string logged to a logger named `'py.warnings'` with a severity of `WARNING`.

If *capture* is `False`, the redirection of warnings to the logging system will stop, and warnings will be redirected to their original destinations (i.e. those in effect before `captureWarnings(True)` was called).

Voir aussi

Module `logging.config`
API de configuration pour le module de journalisation.

Module `logging.handlers`
Gestionnaires utiles inclus avec le module de journalisation.

PEP 282 - A Logging System
The proposal which described this feature for inclusion in the Python standard library.

Original Python logging package
This is the original source for the `logging` package. The version of the package available from this site is suitable for use with Python 1.5.2, 2.1.x and 2.2.x, which do not include the `logging` package in the standard library.

16.5 `logging.config` --- Logging configuration

Source code : Lib/logging/config.py

Important

Cette page contient uniquement des informations de référence. Pour des tutoriels, veuillez consulter
- Tutoriel basique
- Tutoriel avancé
- Recettes pour la journalisation

This section describes the API for configuring the logging module.

16.5.1 Configuration functions

The following functions configure the logging module. They are located in the `logging.config` module. Their use is optional --- you can configure the logging module using these functions or by making calls to the main API (defined in `logging` itself) and defining handlers which are declared either in `logging` or `logging.handlers`.

`logging.config.dictConfig`(*config*)

Takes the logging configuration from a dictionary. The contents of this dictionary are described in *Configuration dictionary schema* below.

If an error is encountered during configuration, this function will raise a *ValueError*, *TypeError*, *AttributeError* or *ImportError* with a suitably descriptive message. The following is a (possibly incomplete) list of conditions which will raise an error :

— A `level` which is not a string or which is a string not corresponding to an actual logging level.
— A `propagate` value which is not a boolean.
— An id which does not have a corresponding destination.
— A non-existent handler id found during an incremental call.
— An invalid logger name.
— Inability to resolve to an internal or external object.

Parsing is performed by the `DictConfigurator` class, whose constructor is passed the dictionary used for configuration, and has a `configure()` method. The *logging.config* module has a callable attribute `dictConfigClass` which is initially set to `DictConfigurator`. You can replace the value of `dictConfigClass` with a suitable implementation of your own.

dictConfig() calls `dictConfigClass` passing the specified dictionary, and then calls the `configure()` method on the returned object to put the configuration into effect :

```
def dictConfig(config):
    dictConfigClass(config).configure()
```

For example, a subclass of `DictConfigurator` could call `DictConfigurator.__init__()` in its own `__init__()`, then set up custom prefixes which would be usable in the subsequent `configure()` call. `dictConfigClass` would be bound to this new subclass, and then *dictConfig()* could be called exactly as in the default, uncustomized state.

Ajouté dans la version 3.2.

`logging.config.`**`fileConfig`**(*fname*, *defaults=None*, *disable_existing_loggers=True*, *encoding=None*)

Reads the logging configuration from a *configparser*-format file. The format of the file should be as described in *Configuration file format*. This function can be called several times from an application, allowing an end user to select from various pre-canned configurations (if the developer provides a mechanism to present the choices and load the chosen configuration).

It will raise *FileNotFoundError* if the file doesn't exist and *RuntimeError* if the file is invalid or empty.

Paramètres

— **fname** -- A filename, or a file-like object, or an instance derived from *RawConfigParser*. If a `RawConfigParser`-derived instance is passed, it is used as is. Otherwise, a *ConfigParser* is instantiated, and the configuration read by it from the object passed in `fname`. If that has a *readline()* method, it is assumed to be a file-like object and read using *read_file()* ; otherwise, it is assumed to be a filename and passed to *read()*.
— **defaults** -- Defaults to be passed to the `ConfigParser` can be specified in this argument.
— **disable_existing_loggers** -- If specified as `False`, loggers which exist when this call is made are left enabled. The default is `True` because this enables old behaviour in a backward-compatible way. This behaviour is to disable any existing non-root loggers unless they or their ancestors are explicitly named in the logging configuration.
— **encoding** -- The encoding used to open file when *fname* is filename.

Modifié dans la version 3.4 : An instance of a subclass of *RawConfigParser* is now accepted as a value for `fname`. This facilitates :

— Use of a configuration file where logging configuration is just part of the overall application configuration.
— Use of a configuration read from a file, and then modified by the using application (e.g. based on command-line parameters or other aspects of the runtime environment) before being passed to `fileConfig`.

Modifié dans la version 3.10 : Added the *encoding* parameter.

Modifié dans la version 3.12 : An exception will be thrown if the provided file doesn't exist or is invalid or empty.

`logging.config.`**`listen`**(*port=DEFAULT_LOGGING_CONFIG_PORT*, *verify=None*)

Starts up a socket server on the specified port, and listens for new configurations. If no port is specified, the module's default `DEFAULT_LOGGING_CONFIG_PORT` is used. Logging configurations will be sent as a file suitable for processing by *dictConfig()* or *fileConfig()*. Returns a *Thread* instance on which you

can call `start()` to start the server, and which you can `join()` when appropriate. To stop the server, call `stopListening()`.

The `verify` argument, if specified, should be a callable which should verify whether bytes received across the socket are valid and should be processed. This could be done by encrypting and/or signing what is sent across the socket, such that the `verify` callable can perform signature verification and/or decryption. The `verify` callable is called with a single argument - the bytes received across the socket - and should return the bytes to be processed, or `None` to indicate that the bytes should be discarded. The returned bytes could be the same as the passed in bytes (e.g. when only verification is done), or they could be completely different (perhaps if decryption were performed).

To send a configuration to the socket, read in the configuration file and send it to the socket as a sequence of bytes preceded by a four-byte length string packed in binary using `struct.pack('>L', n)`.

> **Note**
>
> Because portions of the configuration are passed through `eval()`, use of this function may open its users to a security risk. While the function only binds to a socket on `localhost`, and so does not accept connections from remote machines, there are scenarios where untrusted code could be run under the account of the process which calls `listen()`. Specifically, if the process calling `listen()` runs on a multi-user machine where users cannot trust each other, then a malicious user could arrange to run essentially arbitrary code in a victim user's process, simply by connecting to the victim's `listen()` socket and sending a configuration which runs whatever code the attacker wants to have executed in the victim's process. This is especially easy to do if the default port is used, but not hard even if a different port is used. To avoid the risk of this happening, use the `verify` argument to `listen()` to prevent unrecognised configurations from being applied.

Modifié dans la version 3.4 : The `verify` argument was added.

> **Note**
>
> If you want to send configurations to the listener which don't disable existing loggers, you will need to use a JSON format for the configuration, which will use `dictConfig()` for configuration. This method allows you to specify `disable_existing_loggers` as `False` in the configuration you send.

`logging.config.`**`stopListening`**`()`

Stops the listening server which was created with a call to `listen()`. This is typically called before calling `join()` on the return value from `listen()`.

16.5.2 Security considerations

The logging configuration functionality tries to offer convenience, and in part this is done by offering the ability to convert text in configuration files into Python objects used in logging configuration - for example, as described in *User-defined objects*. However, these same mechanisms (importing callables from user-defined modules and calling them with parameters from the configuration) could be used to invoke any code you like, and for this reason you should treat configuration files from untrusted sources with *extreme caution* and satisfy yourself that nothing bad can happen if you load them, before actually loading them.

16.5.3 Configuration dictionary schema

Describing a logging configuration requires listing the various objects to create and the connections between them ; for example, you may create a handler named 'console' and then say that the logger named 'startup' will send its messages to the 'console' handler. These objects aren't limited to those provided by the `logging` module because you might write your own formatter or handler class. The parameters to these classes may also need to include external objects such as `sys.stderr`. The syntax for describing these objects and connections is defined in *Object connections* below.

Dictionary Schema Details

The dictionary passed to `dictConfig()` must contain the following keys :

— *version* - to be set to an integer value representing the schema version. The only valid value at present is 1, but having this key allows the schema to evolve while still preserving backwards compatibility.

All other keys are optional, but if present they will be interpreted as described below. In all cases below where a 'configuring dict' is mentioned, it will be checked for the special `'()'` key to see if a custom instantiation is required. If so, the mechanism described in *User-defined objects* below is used to create an instance ; otherwise, the context is used to determine what to instantiate.

— *formatters* - the corresponding value will be a dict in which each key is a formatter id and each value is a dict describing how to configure the corresponding `Formatter` instance.

The configuring dict is searched for the following optional keys which correspond to the arguments passed to create a `Formatter` object :

— `format`
— `datefmt`
— `style`
— `validate` (since version >=3.8)
— `defaults` (since version >=3.12)

An optional `class` key indicates the name of the formatter's class (as a dotted module and class name). The instantiation arguments are as for `Formatter`, thus this key is most useful for instantiating a customised subclass of `Formatter`. For example, the alternative class might present exception tracebacks in an expanded or condensed format. If your formatter requires different or extra configuration keys, you should use *User-defined objects*.

— *filters* - the corresponding value will be a dict in which each key is a filter id and each value is a dict describing how to configure the corresponding Filter instance.

The configuring dict is searched for the key `name` (defaulting to the empty string) and this is used to construct a `logging.Filter` instance.

— *handlers* - the corresponding value will be a dict in which each key is a handler id and each value is a dict describing how to configure the corresponding Handler instance.

The configuring dict is searched for the following keys :

— `class` (mandatory). This is the fully qualified name of the handler class.
— `level` (optional). The level of the handler.
— `formatter` (optional). The id of the formatter for this handler.
— `filters` (optional). A list of ids of the filters for this handler.

Modifié dans la version 3.11 : `filters` can take filter instances in addition to ids.

All *other* keys are passed through as keyword arguments to the handler's constructor. For example, given the snippet :

```
handlers:
  console:
    class : logging.StreamHandler
    formatter: brief
    level   : INFO
    filters: [allow_foo]
    stream  : ext://sys.stdout
  file:
    class : logging.handlers.RotatingFileHandler
    formatter: precise
    filename: logconfig.log
    maxBytes: 1024
    backupCount: 3
```

the handler with id `console` is instantiated as a `logging.StreamHandler`, using `sys.stdout` as the underlying stream. The handler with id `file` is instantiated as a `logging.handlers.RotatingFileHandler` with the keyword arguments `filename='logconfig.log', maxBytes=1024, backupCount=3`.

— *loggers* - the corresponding value will be a dict in which each key is a logger name and each value is a dict describing how to configure the corresponding Logger instance.

  The configuring dict is searched for the following keys :
  — `level` (optional). The level of the logger.
  — `propagate` (optional). The propagation setting of the logger.
  — `filters` (optional). A list of ids of the filters for this logger.

    Modifié dans la version 3.11 : `filters` can take filter instances in addition to ids.
  — `handlers` (optional). A list of ids of the handlers for this logger.
  The specified loggers will be configured according to the level, propagation, filters and handlers specified.
— *root* - this will be the configuration for the root logger. Processing of the configuration will be as for any logger, except that the `propagate` setting will not be applicable.
— *incremental* - whether the configuration is to be interpreted as incremental to the existing configuration. This value defaults to `False`, which means that the specified configuration replaces the existing configuration with the same semantics as used by the existing `fileConfig()` API.

  If the specified value is `True`, the configuration is processed as described in the section on *Incremental Configuration*.
— *disable_existing_loggers* - whether any existing non-root loggers are to be disabled. This setting mirrors the parameter of the same name in `fileConfig()`. If absent, this parameter defaults to `True`. This value is ignored if *incremental* is `True`.

Incremental Configuration

It is difficult to provide complete flexibility for incremental configuration. For example, because objects such as filters and formatters are anonymous, once a configuration is set up, it is not possible to refer to such anonymous objects when augmenting a configuration.

Furthermore, there is not a compelling case for arbitrarily altering the object graph of loggers, handlers, filters, formatters at run-time, once a configuration is set up ; the verbosity of loggers and handlers can be controlled just by setting levels (and, in the case of loggers, propagation flags). Changing the object graph arbitrarily in a safe way is problematic in a multi-threaded environment ; while not impossible, the benefits are not worth the complexity it adds to the implementation.

Thus, when the `incremental` key of a configuration dict is present and is `True`, the system will completely ignore any `formatters` and `filters` entries, and process only the `level` settings in the `handlers` entries, and the `level` and `propagate` settings in the `loggers` and `root` entries.

Using a value in the configuration dict lets configurations to be sent over the wire as pickled dicts to a socket listener. Thus, the logging verbosity of a long-running application can be altered over time with no need to stop and restart the application.

Object connections

The schema describes a set of logging objects - loggers, handlers, formatters, filters - which are connected to each other in an object graph. Thus, the schema needs to represent connections between the objects. For example, say that, once configured, a particular logger has attached to it a particular handler. For the purposes of this discussion, we can say that the logger represents the source, and the handler the destination, of a connection between the two. Of course in the configured objects this is represented by the logger holding a reference to the handler. In the configuration dict, this is done by giving each destination object an id which identifies it unambiguously, and then using the id in the source object's configuration to indicate that a connection exists between the source and the destination object with that id.

So, for example, consider the following YAML snippet :

```
formatters:
  brief:
    # configuration for formatter with id 'brief' goes here
  precise:
    # configuration for formatter with id 'precise' goes here
```

(suite sur la page suivante)

(suite de la page précédente)

```
handlers:
  h1: #This is an id
   # configuration of handler with id 'h1' goes here
   formatter: brief
  h2: #This is another id
   # configuration of handler with id 'h2' goes here
   formatter: precise
loggers:
  foo.bar.baz:
    # other configuration for logger 'foo.bar.baz'
    handlers: [h1, h2]
```

(Note : YAML used here because it's a little more readable than the equivalent Python source form for the dictionary.)

The ids for loggers are the logger names which would be used programmatically to obtain a reference to those loggers, e.g. `foo.bar.baz`. The ids for Formatters and Filters can be any string value (such as `brief`, `precise` above) and they are transient, in that they are only meaningful for processing the configuration dictionary and used to determine connections between objects, and are not persisted anywhere when the configuration call is complete.

The above snippet indicates that logger named `foo.bar.baz` should have two handlers attached to it, which are described by the handler ids `h1` and `h2`. The formatter for `h1` is that described by id `brief`, and the formatter for `h2` is that described by id `precise`.

User-defined objects

The schema supports user-defined objects for handlers, filters and formatters. (Loggers do not need to have different types for different instances, so there is no support in this configuration schema for user-defined logger classes.)

Objects to be configured are described by dictionaries which detail their configuration. In some places, the logging system will be able to infer from the context how an object is to be instantiated, but when a user-defined object is to be instantiated, the system will not know how to do this. In order to provide complete flexibility for user-defined object instantiation, the user needs to provide a 'factory' - a callable which is called with a configuration dictionary and which returns the instantiated object. This is signalled by an absolute import path to the factory being made available under the special key `'()'`. Here's a concrete example :

```
formatters:
  brief:
    format: '%(message)s'
  default:
    format: '%(asctime)s %(levelname)-8s %(name)-15s %(message)s'
    datefmt: '%Y-%m-%d %H:%M:%S'
  custom:
      (): my.package.customFormatterFactory
      bar: baz
      spam: 99.9
      answer: 42
```

The above YAML snippet defines three formatters. The first, with id `brief`, is a standard *`logging.Formatter`* instance with the specified format string. The second, with id `default`, has a longer format and also defines the time format explicitly, and will result in a *`logging.Formatter`* initialized with those two format strings. Shown in Python source form, the `brief` and `default` formatters have configuration sub-dictionaries :

```
{
  'format' : '%(message)s'
}
```

et :

```
{
  'format' : '%(asctime)s %(levelname)-8s %(name)-15s %(message)s',
  'datefmt' : '%Y-%m-%d %H:%M:%S'
}
```

respectively, and as these dictionaries do not contain the special key `'()'`, the instantiation is inferred from the context : as a result, standard *logging.Formatter* instances are created. The configuration sub-dictionary for the third formatter, with id `custom`, is :

```
{
  '()' : 'my.package.customFormatterFactory',
  'bar' : 'baz',
  'spam' : 99.9,
  'answer' : 42
}
```

and this contains the special key `'()'`, which means that user-defined instantiation is wanted. In this case, the specified factory callable will be used. If it is an actual callable it will be used directly - otherwise, if you specify a string (as in the example) the actual callable will be located using normal import mechanisms. The callable will be called with the **remaining** items in the configuration sub-dictionary as keyword arguments. In the above example, the formatter with id `custom` will be assumed to be returned by the call :

```
my.package.customFormatterFactory(bar='baz', spam=99.9, answer=42)
```

Avertissement

The values for keys such as `bar`, `spam` and `answer` in the above example should not be configuration dictionaries or references such as `cfg://foo` or `ext://bar`, because they will not be processed by the configuration machinery, but passed to the callable as-is.

The key `'()'` has been used as the special key because it is not a valid keyword parameter name, and so will not clash with the names of the keyword arguments used in the call. The `'()'` also serves as a mnemonic that the corresponding value is a callable.

Modifié dans la version 3.11 : The `filters` member of `handlers` and `loggers` can take filter instances in addition to ids.

You can also specify a special key `'.'` whose value is a dictionary is a mapping of attribute names to values. If found, the specified attributes will be set on the user-defined object before it is returned. Thus, with the following configuration :

```
{
  '()' : 'my.package.customFormatterFactory',
  'bar' : 'baz',
  'spam' : 99.9,
  'answer' : 42,
  '.' {
    'foo': 'bar',
    'baz': 'bozz'
  }
}
```

the returned formatter will have attribute `foo` set to `'bar'` and attribute `baz` set to `'bozz'`.

Avertissement

The values for attributes such as `foo` and `baz` in the above example should not be configuration dictionaries or references such as `cfg://foo` or `ext://bar`, because they will not be processed by the configuration machinery, but set as attribute values as-is.

Handler configuration order

Handlers are configured in alphabetical order of their keys, and a configured handler replaces the configuration dictionary in (a working copy of) the `handlers` dictionary in the schema. If you use a construct such as `cfg://handlers.foo`, then initially `handlers['foo']` points to the configuration dictionary for the handler named `foo`, and later (once that handler has been configured) it points to the configured handler instance. Thus, `cfg://handlers.foo` could resolve to either a dictionary or a handler instance. In general, it is wise to name handlers in a way such that dependent handlers are configured _after_ any handlers they depend on ; that allows something like `cfg://handlers.foo` to be used in configuring a handler that depends on handler `foo`. If that dependent handler were named `bar`, problems would result, because the configuration of `bar` would be attempted before that of `foo`, and `foo` would not yet have been configured. However, if the dependent handler were named `foobar`, it would be configured after `foo`, with the result that `cfg://handlers.foo` would resolve to configured handler `foo`, and not its configuration dictionary.

Access to external objects

There are times where a configuration needs to refer to objects external to the configuration, for example `sys.stderr`. If the configuration dict is constructed using Python code, this is straightforward, but a problem arises when the configuration is provided via a text file (e.g. JSON, YAML). In a text file, there is no standard way to distinguish `sys.stderr` from the literal string `'sys.stderr'`. To facilitate this distinction, the configuration system looks for certain special prefixes in string values and treat them specially. For example, if the literal string `'ext://sys.stderr'` is provided as a value in the configuration, then the `ext://` will be stripped off and the remainder of the value processed using normal import mechanisms.

The handling of such prefixes is done in a way analogous to protocol handling : there is a generic mechanism to look for prefixes which match the regular expression `^(?P<prefix>[a-z]+)://(?P<suffix>.*)$` whereby, if the `prefix` is recognised, the `suffix` is processed in a prefix-dependent manner and the result of the processing replaces the string value. If the prefix is not recognised, then the string value will be left as-is.

Access to internal objects

As well as external objects, there is sometimes also a need to refer to objects in the configuration. This will be done implicitly by the configuration system for things that it knows about. For example, the string value `'DEBUG'` for a `level` in a logger or handler will automatically be converted to the value `logging.DEBUG`, and the `handlers`, `filters` and `formatter` entries will take an object id and resolve to the appropriate destination object.

However, a more generic mechanism is needed for user-defined objects which are not known to the *logging* module. For example, consider *logging.handlers.MemoryHandler*, which takes a `target` argument which is another handler to delegate to. Since the system already knows about this class, then in the configuration, the given `target` just needs to be the object id of the relevant target handler, and the system will resolve to the handler from the id. If, however, a user defines a `my.package.MyHandler` which has an `alternate` handler, the configuration system would not know that the `alternate` referred to a handler. To cater for this, a generic resolution system allows the user to specify :

```
handlers:
  file:
    # configuration of file handler goes here

  custom:
    (): my.package.MyHandler
    alternate: cfg://handlers.file
```

The literal string `'cfg://handlers.file'` will be resolved in an analogous way to strings with the `ext://` prefix, but looking in the configuration itself rather than the import namespace. The mechanism allows access by dot or by index, in a similar way to that provided by `str.format`. Thus, given the following snippet :

```
handlers:
  email:
    class: logging.handlers.SMTPHandler
    mailhost: localhost
    fromaddr: my_app@domain.tld
    toaddrs:
      - support_team@domain.tld
      - dev_team@domain.tld
    subject: Houston, we have a problem.
```

in the configuration, the string `'cfg://handlers'` would resolve to the dict with key `handlers`, the string `'cfg://handlers.email` would resolve to the dict with key `email` in the `handlers` dict, and so on. The string `'cfg://handlers.email.toaddrs[1]` would resolve to `'dev_team@domain.tld'` and the string `'cfg://handlers.email.toaddrs[0]'` would resolve to the value `'support_team@domain.tld'`. The `subject` value could be accessed using either `'cfg://handlers.email.subject'` or, equivalently, `'cfg://handlers.email[subject]'`. The latter form only needs to be used if the key contains spaces or non-alphanumeric characters. Please note that the characters `[` and `]` are not allowed in the keys. If an index value consists only of decimal digits, access will be attempted using the corresponding integer value, falling back to the string value if needed.

Given a string `cfg://handlers.myhandler.mykey.123`, this will resolve to `config_dict['handlers']['myhandler']['mykey']['123']`. If the string is specified as `cfg://handlers.myhandler.mykey[123]`, the system will attempt to retrieve the value from `config_dict['handlers']['myhandler']['mykey'][123]`, and fall back to `config_dict['handlers']['myhandler']['mykey']['123']` if that fails.

Import resolution and custom importers

Import resolution, by default, uses the builtin *`__import__()`* function to do its importing. You may want to replace this with your own importing mechanism : if so, you can replace the `importer` attribute of the `DictConfigurator` or its superclass, the `BaseConfigurator` class. However, you need to be careful because of the way functions are accessed from classes via descriptors. If you are using a Python callable to do your imports, and you want to define it at class level rather than instance level, you need to wrap it with *`staticmethod()`*. For example :

```
from importlib import import_module
from logging.config import BaseConfigurator

BaseConfigurator.importer = staticmethod(import_module)
```

You don't need to wrap with *`staticmethod()`* if you're setting the import callable on a configurator *instance*.

Configuring QueueHandler and QueueListener

If you want to configure a *`QueueHandler`*, noting that this is normally used in conjunction with a *`QueueListener`*, you can configure both together. After the configuration, the `QueueListener` instance will be available as the *`listener`* attribute of the created handler, and that in turn will be available to you using *`getHandlerByName()`* and passing the name you have used for the `QueueHandler` in your configuration. The dictionary schema for configuring the pair is shown in the example YAML snippet below.

```
handlers:
  qhand:
    class: logging.handlers.QueueHandler
    queue: my.module.queue_factory
    listener: my.package.CustomListener
    handlers:
      - hand_name_1
      - hand_name_2
      ...
```

The `queue` and `listener` keys are optional.

If the `queue` key is present, the corresponding value can be one of the following :

— An object implementing the `Queue.put_nowait` and `Queue.get` public API. For instance, this may be an actual instance of `queue.Queue` or a subclass thereof, or a proxy obtained by `multiprocessing.managers.SyncManager.Queue()`.

This is of course only possible if you are constructing or modifying the configuration dictionary in code.

— A string that resolves to a callable which, when called with no arguments, returns the queue instance to use. That callable could be a `queue.Queue` subclass or a function which returns a suitable queue instance, such as `my.module.queue_factory()`.

— A dict with a `'()'` key which is constructed in the usual way as discussed in *User-defined objects*. The result of this construction should be a `queue.Queue` instance.

If the `queue` key is absent, a standard unbounded `queue.Queue` instance is created and used.

If the `listener` key is present, the corresponding value can be one of the following :

— A subclass of `logging.handlers.QueueListener`. This is of course only possible if you are constructing or modifying the configuration dictionary in code.

— A string which resolves to a class which is a subclass of `QueueListener`, such as `'my.package.CustomListener'`.

— A dict with a `'()'` key which is constructed in the usual way as discussed in *User-defined objects*. The result of this construction should be a callable with the same signature as the `QueueListener` initializer.

If the `listener` key is absent, `logging.handlers.QueueListener` is used.

The values under the `handlers` key are the names of other handlers in the configuration (not shown in the above snippet) which will be passed to the queue listener.

Any custom queue handler and listener classes will need to be defined with the same initialization signatures as `QueueHandler` and `QueueListener`.

Ajouté dans la version 3.12.

16.5.4 Configuration file format

The configuration file format understood by `fileConfig()` is based on `configparser` functionality. The file must contain sections called `[loggers]`, `[handlers]` and `[formatters]` which identify by name the entities of each type which are defined in the file. For each such entity, there is a separate section which identifies how that entity is configured. Thus, for a logger named `log01` in the `[loggers]` section, the relevant configuration details are held in a section `[logger_log01]`. Similarly, a handler called `hand01` in the `[handlers]` section will have its configuration held in a section called `[handler_hand01]`, while a formatter called `form01` in the `[formatters]` section will have its configuration specified in a section called `[formatter_form01]`. The root logger configuration must be specified in a section called `[logger_root]`.

Note

The `fileConfig()` API is older than the `dictConfig()` API and does not provide functionality to cover certain aspects of logging. For example, you cannot configure `Filter` objects, which provide for filtering of messages beyond simple integer levels, using `fileConfig()`. If you need to have instances of `Filter` in your logging configuration, you will need to use `dictConfig()`. Note that future enhancements to configuration functionality will be added to `dictConfig()`, so it's worth considering transitioning to this newer API when it's convenient to do so.

Examples of these sections in the file are given below.

```
[loggers]
keys=root,log02,log03,log04,log05,log06,log07

[handlers]
keys=hand01,hand02,hand03,hand04,hand05,hand06,hand07,hand08,hand09

[formatters]
keys=form01,form02,form03,form04,form05,form06,form07,form08,form09
```

The root logger must specify a level and a list of handlers. An example of a root logger section is given below.

```
[logger_root]
level=NOTSET
handlers=hand01
```

The `level` entry can be one of `DEBUG, INFO, WARNING, ERROR, CRITICAL` or `NOTSET`. For the root logger only, `NOTSET` means that all messages will be logged. Level values are *evaluated* in the context of the `logging` package's namespace.

The `handlers` entry is a comma-separated list of handler names, which must appear in the `[handlers]` section. These names must appear in the `[handlers]` section and have corresponding sections in the configuration file.

For loggers other than the root logger, some additional information is required. This is illustrated by the following example.

```
[logger_parser]
level=DEBUG
handlers=hand01
propagate=1
qualname=compiler.parser
```

The `level` and `handlers` entries are interpreted as for the root logger, except that if a non-root logger's level is specified as `NOTSET`, the system consults loggers higher up the hierarchy to determine the effective level of the logger. The `propagate` entry is set to 1 to indicate that messages must propagate to handlers higher up the logger hierarchy from this logger, or 0 to indicate that messages are **not** propagated to handlers up the hierarchy. The `qualname` entry is the hierarchical channel name of the logger, that is to say the name used by the application to get the logger.

Sections which specify handler configuration are exemplified by the following.

```
[handler_hand01]
class=StreamHandler
level=NOTSET
formatter=form01
args=(sys.stdout,)
```

The `class` entry indicates the handler's class (as determined by `eval()` in the `logging` package's namespace). The `level` is interpreted as for loggers, and `NOTSET` is taken to mean 'log everything'.

The `formatter` entry indicates the key name of the formatter for this handler. If blank, a default formatter (`logging._defaultFormatter`) is used. If a name is specified, it must appear in the `[formatters]` section and have a corresponding section in the configuration file.

The `args` entry, when *evaluated* in the context of the `logging` package's namespace, is the list of arguments to the constructor for the handler class. Refer to the constructors for the relevant handlers, or to the examples below, to see how typical entries are constructed. If not provided, it defaults to `()`.

The optional `kwargs` entry, when *evaluated* in the context of the `logging` package's namespace, is the keyword argument dict to the constructor for the handler class. If not provided, it defaults to `{}`.

```
[handler_hand02]
class=FileHandler
level=DEBUG
formatter=form02
args=('python.log', 'w')

[handler_hand03]
class=handlers.SocketHandler
level=INFO
formatter=form03
args=('localhost', handlers.DEFAULT_TCP_LOGGING_PORT)
```

(suite sur la page suivante)

(suite de la page précédente)

```
[handler_hand04]
class=handlers.DatagramHandler
level=WARN
formatter=form04
args=('localhost', handlers.DEFAULT_UDP_LOGGING_PORT)

[handler_hand05]
class=handlers.SysLogHandler
level=ERROR
formatter=form05
args=(('localhost', handlers.SYSLOG_UDP_PORT), handlers.SysLogHandler.LOG_USER)

[handler_hand06]
class=handlers.NTEventLogHandler
level=CRITICAL
formatter=form06
args=('Python Application', '', 'Application')

[handler_hand07]
class=handlers.SMTPHandler
level=WARN
formatter=form07
args=('localhost', 'from@abc', ['user1@abc', 'user2@xyz'], 'Logger Subject')
kwargs={'timeout': 10.0}

[handler_hand08]
class=handlers.MemoryHandler
level=NOTSET
formatter=form08
target=
args=(10, ERROR)

[handler_hand09]
class=handlers.HTTPHandler
level=NOTSET
formatter=form09
args=('localhost:9022', '/log', 'GET')
kwargs={'secure': True}
```

Sections which specify formatter configuration are typified by the following.

```
[formatter_form01]
format=F1 %(asctime)s %(levelname)s %(message)s %(customfield)s
datefmt=
style=%
validate=True
defaults={'customfield': 'defaultvalue'}
class=logging.Formatter
```

The arguments for the formatter configuration are the same as the keys in the dictionary schema *formatters section*.

The `defaults` entry, when *evaluated* in the context of the `logging` package's namespace, is a dictionary of default values for custom formatting fields. If not provided, it defaults to `None`.

Note

Due to the use of `eval()` as described above, there are potential security risks which result from using the `listen()` to send and receive configurations via sockets. The risks are limited to where multiple users with no mutual trust run code on the same machine ; see the `listen()` documentation for more information.

Voir aussi

Module `logging`
Référence d'API pour le module de journalisation.

Module `logging.handlers`
Gestionnaires utiles inclus avec le module de journalisation.

16.6 `logging.handlers` --- Logging handlers

Code source : Lib/logging/handlers.py

Important

Cette page contient uniquement des informations de référence. Pour des tutoriels, veuillez consulter
- Tutoriel basique
- Tutoriel avancé
- livre de recettes sur la journalisation

Les gestionnaires suivants, très utiles, sont fournis dans le paquet. Notez que trois des gestionnaires (`StreamHandler`, `FileHandler` et `NullHandler`) sont en réalité définis dans le module `logging` lui-même, mais qu'ils sont documentés ici avec les autres gestionnaires.

16.6.1 Gestionnaire à flux — *StreamHandler*

La classe `StreamHandler`, du paquet `logging`, envoie les sorties de journalisation dans des flux tels que *sys.stdout*, *sys.stderr* ou n'importe quel objet fichier-compatible (ou, plus précisément, tout objet qui gère les méthodes `write()` et `flush()`).

class `logging.`**`StreamHandler`**(*stream=None*)

Renvoie une nouvelle instance de la classe `StreamHandler`. Si *stream* est spécifié, l'instance l'utilise pour les sorties de journalisation ; autrement elle utilise *sys.stderr*.

emit(*record*)

Si un formateur est spécifié, il est utilisé pour formater l'enregistrement. L'enregistrement est ensuite écrit dans le flux, suivi par `terminator`. Si une information d'exception est présente, elle est formatée en utilisant `traceback.print_exception()` puis ajoutée aux flux.

flush()

Purge le flux en appelant sa méthode `flush()`. Notez que la méthode `close()` est héritée de `Handler` donc elle n'écrit rien. Par conséquent, un appel explicite à `flush()` peut parfois s'avérer nécessaire.

setStream(*stream*)

Définit le flux de l'instance à la valeur spécifiée, si elle est différente. L'ancien flux est purgé avant que le nouveau flux ne soit établi.

Paramètres
stream -- le flux que le gestionnaire doit utiliser.

Renvoie
the old stream, if the stream was changed, or `None` if it wasn't.

Ajouté dans la version 3.7.

terminator

Chaine de caractères utilisée comme marqueur de fin lors de l'écriture formatée d'un enregistrement dans un flux. La valeur par défaut est `'\n'`.

Si vous ne voulez pas marquer de fin de ligne, il faut définir l'attribut `terminator` à la chaîne de caractères vide.

Dans des versions antérieures, le marqueur de fin était codé en dur sous la forme `'\n'`.

Ajouté dans la version 3.2.

16.6.2 Gestionnaire à fichier — *FileHandler*

La classe `FileHandler`, du paquet `logging`, envoie les sorties de journalisation dans un fichier. Elle hérite des fonctionnalités de sortie de `StreamHandler`.

class `logging.`**FileHandler**(*filename*, *mode='a'*, *encoding=None*, *delay=False*, *errors=None*)

Returns a new instance of the `FileHandler` class. The specified file is opened and used as the stream for logging. If *mode* is not specified, `'a'` is used. If *encoding* is not `None`, it is used to open the file with that encoding. If *delay* is true, then file opening is deferred until the first call to `emit()`. By default, the file grows indefinitely. If *errors* is specified, it's used to determine how encoding errors are handled.

Modifié dans la version 3.6 : L'argument *filename* accepte les objets `Path` aussi bien que les chaînes de caractères.

Modifié dans la version 3.9 : ajout du paramètre *errors*.

close()

Ferme le fichier.

emit(*record*)

Écrit l'enregistrement dans le fichier.

Note that if the file was closed due to logging shutdown at exit and the file mode is 'w', the record will not be emitted (see bpo-42378).

16.6.3 Gestionnaire à puits sans fond — *NullHandler*

Ajouté dans la version 3.1.

La classe `NullHandler`, située dans le paquet principal `logging`, ne produit aucun formatage ni sortie. C'est essentiellement un gestionnaire « fantôme » destiné aux développeurs de bibliothèques.

class `logging.`**NullHandler**

Renvoie une nouvelle instance de la classe `NullHandler`.

emit(*record*)

Cette méthode ne fait rien.

handle(*record*)

Cette méthode ne fait rien.

createLock()

Cette méthode renvoie `None` pour le verrou, étant donné qu'il n'y a aucun flux d'entrée-sortie sous-jacent dont l'accès doit être sérialisé.

Voir library-config pour plus d'information sur l'utilisation de `NullHandler`.

16.6.4 Gestionnaire à fichier avec surveillance — *WatchedFileHandler*

La classe `WatchedFileHandler`, du module `logging.handlers`, est une classe `FileHandler` qui surveille le fichier dans lequel elle écrit. Si le fichier est modifié, il est fermé et rouvert en utilisant le nom du fichier.

Le fichier peut être modifié par des programmes tels que *newsyslog* ou *logrotate* qui assurent le roulement des fichiers de journalisation. Ce gestionnaire, destiné à une utilisation sous Unix-Linux, surveille le fichier pour voir s'il a changé depuis la dernière écriture (un fichier est considéré comme modifié si son nœud d'index ou le périphérique auquel il est rattaché a changé). Si le fichier a changé, l'ancien flux vers ce fichier est fermé, et le fichier est ouvert pour établir un nouveau flux.

Ce gestionnaire n'est pas approprié pour une utilisation sous *Windows*, car sous *Windows* les fichiers de journalisation ouverts ne peuvent être ni déplacés, ni renommés — la journalisation ouvre les fichiers avec des verrous exclusifs — de telle sorte qu'il n'y a pas besoin d'un tel gestionnaire. En outre, *ST_INO* n'est pas géré par *Windows* ; `stat()` renvoie toujours zéro pour cette valeur.

class `logging.handlers.`**`WatchedFileHandler`**(*filename*, *mode='a'*, *encoding=None*, *delay=False*, *errors=None*)

Returns a new instance of the `WatchedFileHandler` class. The specified file is opened and used as the stream for logging. If *mode* is not specified, `'a'` is used. If *encoding* is not `None`, it is used to open the file with that encoding. If *delay* is true, then file opening is deferred until the first call to `emit()`. By default, the file grows indefinitely. If *errors* is provided, it determines how encoding errors are handled.

Modifié dans la version 3.6 : L'argument *filename* accepte les objets `Path` aussi bien que les chaînes de caractères.

Modifié dans la version 3.9 : ajout du paramètre *errors*.

reopenIfNeeded()

Vérifie si le fichier a changé. Si c'est le cas, le flux existant est purgé et fermé et le fichier est rouvert, généralement avant d'effectuer l'écriture de l'enregistrement dans le fichier.

Ajouté dans la version 3.6.

emit(*record*)

Écrit l'enregistrement dans le fichier, mais appelle d'abord `reopenIfNeeded()` pour rouvrir le fichier s'il a changé.

16.6.5 Base des gestionnaires à roulement — *BaseRotatingHandler*

La classe `BaseRotatingHandler`, du module `logging.handlers`, est la classe de base pour les gestionnaires à roulement, `RotatingFileHandler` et `TimedRotatingFileHandler`. Il ne faut pas créer des instances de cette classe, mais surcharger ses attributs et méthodes.

class `logging.handlers.`**`BaseRotatingHandler`**(*filename*, *mode*, *encoding=None*, *delay=False*, *errors=None*)

Les paramètres sont les mêmes que pour `FileHandler`. Les attributs sont :

namer

Si cet attribut est un qu'appelable, la méthode `rotation_filename()` délègue sa logique à cet appelable. Les paramètres passés à l'appelable sont ceux passés à `rotation_filename()`.

> **Note**
>
> la fonction *namer* est appelée de nombreuses fois durant le roulement ; elle doit donc être aussi simple et rapide que possible. Elle doit aussi renvoyer toujours la même sortie pour une entrée donnée, autrement le comportement du roulement pourrait être différent de celui attendu.
>
> It's also worth noting that care should be taken when using a namer to preserve certain attributes in the filename which are used during rotation. For example, `RotatingFileHandler` expects to have a set of log files whose names contain successive integers, so that rotation works as expected, and `TimedRotatingFileHandler` deletes old log files (based on the `backupCount` parameter

passed to the handler's initializer) by determining the oldest files to delete. For this to happen, the filenames should be sortable using the date/time portion of the filename, and a namer needs to respect this. (If a namer is wanted that doesn't respect this scheme, it will need to be used in a subclass of `TimedRotatingFileHandler` which overrides the `getFilesToDelete()` method to fit in with the custom naming scheme.)

Ajouté dans la version 3.3.

rotator

Si cet attribut est un appelable, il se substitue à la méthode `rotate()`. Les paramètres passés à l'appelable sont ceux passés à `rotate()`.

Ajouté dans la version 3.3.

rotation_filename(*default_name*)

Modifie le nom du fichier d'un fichier de journalisation lors du roulement.

Cette méthode sert à pouvoir produire un nom de fichier personnalisé.

L'implémentation par défaut appelle l'attribut *namer* du gestionnaire, si c'est un appelable, lui passant le nom par défaut. Si l'attribut n'est pas un appelable (le défaut est `None`), le nom est renvoyé tel quel.

Paramètres

default_name -- le nom par défaut du fichier de journalisation.

Ajouté dans la version 3.3.

rotate(*source*, *dest*)

Lors du roulement, effectue le roulement du journal courant.

L'implémentation par défaut appelle l'attribut *rotator* du gestionnaire, si c'est un appelable, lui passant les arguments *source* et *dest*. Si l'attribut n'est pas un appelable (le défaut est `None`), le nom de la source est simplement renommé avec la destination.

Paramètres

- **source** -- le nom du fichier source. Il s'agit normalement du nom du fichier, par exemple `"test.log"`.
- **dest** -- le nom du fichier de destination. Il s'agit normalement du nom donné à la source après le roulement, par exemple `"test.log.1"`.

Ajouté dans la version 3.3.

La raison d'être de ces attributs est de vous épargner la création d'une sous-classe — vous pouvez utiliser les mêmes appels pour des instances de `RotatingFileHandler` et `TimedRotatingFileHandler`. Si l'appelable *namer* ou *rotator* lève une exception, elle est gérée de la même manière que n'importe quelle exception levée lors d'un appel à `emit()`, c'est-à-dire par la méthode `handleError()` du gestionnaire.

Si vous avez besoin de faire d'importantes modifications au processus de roulement, surchargez les méthodes.

Pour un exemple, voir cookbook-rotator-namer.

16.6.6 Gestionnaire à roulement de fichiers — *RotatingFileHandler*

La classe `RotatingFileHandler`, du module `logging.handlers`, gère le roulement des fichiers de journalisation sur disque.

class logging.handlers.**RotatingFileHandler**(*filename*, *mode='a'*, *maxBytes=0*, *backupCount=0*, *encoding=None*, *delay=False*, *errors=None*)

Renvoie une nouvelle instance de la classe `RotatingFileHandler`. Le fichier spécifié est ouvert et utilisé comme flux de sortie pour la journalisation. Si *mode* n'est pas spécifié, `'a'` est utilisé. Si *encoding* n'est pas à `None`, il est utilisé pour ouvrir le fichier avec cet encodage. Si *delay* est à *true*, alors l'ouverture du fichier est reportée au premier appel de `emit()`. Par défaut, le fichier croit indéfiniment. Si *errors* est spécifié, il détermine comment sont gérées les erreurs d'encodage.

Utilisez les valeurs *maxBytes* et *backupCount* pour autoriser le roulement du fichier (*rollover*) à une taille prédéterminée. Quand la taille limite est sur le point d'être dépassée, le fichier est fermé et un nouveau fichier est automatiquement ouvert pour le remplacer. Un roulement se produit dès que le fichier de journalisation actuel atteint une taille proche de *maxBytes* ; si *maxBytes* ou *backupCount* est à 0, le roulement ne se produit jamais, donc en temps normal il convient de définir *backupCount* à au moins 1, et avoir une valeur de *maxBytes* non nulle. Quand *backupCount* est non nul, le système sauvegarde les anciens fichiers de journalisation en ajoutant à leur nom les suffixes `".1"`, `".2"` et ainsi de suite. Par exemple, avec un *backupCount* de 5 et `app.log` comme radical du fichier, on obtient `app.log`, `app.log.1`, `app.log.2`, jusqu'à `app.log.5`. Le fichier dans lequel on écrit est toujours `app.log`. Quand ce fichier est rempli, il est fermé et renommé en `app.log.1`, et si les fichiers `app.log.1`, `app.log.2`, etc. existent, alors ils sont renommés en `app.log.2`, `app.log.3` etc. respectivement.

Modifié dans la version 3.6 : L'argument *filename* accepte les objets `Path` aussi bien que les chaînes de caractères.

Modifié dans la version 3.9 : ajout du paramètre *errors*.

doRollover()

Effectue un roulement, comme décrit ci-dessus.

emit(*record*)

Écrit l'enregistrement dans le fichier, effectuant un roulement au besoin comme décrit précédemment.

16.6.7 Gestionnaire à roulement de fichiers périodique — *TimedRotatingFileHandler*

La classe `TimedRotatingFileHandler` du module `logging.handlers` gère le roulement des fichiers de journalisation sur le disque selon un intervalle de temps spécifié.

class logging.handlers.**TimedRotatingFileHandler**(*filename*, *when='h'*, *interval=1*, *backupCount=0*, *encoding=None*, *delay=False*, *utc=False*, *atTime=None*, *errors=None*)

Renvoie une nouvelle instance de la classe `TimedRotatingFileHandler`. Le fichier spécifié est ouvert et utilisé comme flux de sortie pour la journalisation. Au moment du roulement, elle met également à jour le suffixe du nom du fichier. L'intervalle entre les roulements est déterminé par les valeurs de *when* et *interval*.

Utilisez *when* pour spécifier le type de *interval*. Ses valeurs possibles sont décrites ci-dessous. Notez qu'elles ne sont pas sensibles à la casse.

Valeur	Type d'intervalle	Rôle de *atTime*
`'S'`	Secondes	Ignoré
`'M'`	Minutes	Ignoré
`'H'`	Heures	Ignoré
`'D'`	Jours	Ignoré
`'W0'-'W6`	Jour de la semaine (0 = lundi)	Utilisé pour calculer le moment du roulement
`'midnigh`	Roulement du fichier à minuit, si *atTime* n'est pas spécifié, sinon à l'heure *atTime*	Utilisé pour calculer le moment du roulement

Pour un roulement selon les jours de la semaine, mettez la valeur *W0* pour lundi, *W1* pour mardi, et ainsi de suite jusqu'à *W6* pour dimanche. Dans ce cas, la valeur indiquée pour *interval* n'est pas utilisée.

Le système sauvegarde les anciens fichiers de journalisation en ajoutant une extension au nom du fichier. Les extensions sont basées sur la date et l'heure, au format *strftime* `%Y-%m-%d_%H-%M-%S` ou le début de celui-ci, selon l'intervalle du roulement.

Le premier calcul de la date du prochain roulement (quand le gestionnaire est créé) dépend de la dernière date de modification d'un fichier de journalisation existant, ou à défaut de la date actuelle.

Si *utc* est vrai, les temps sont UTC, sinon l'heure locale est utilisée.

Si *backupCount* est non nul, au plus *backupCount* fichiers sont sauvegardés. Si des fichiers supplémentaires devaient être créés par le jeu du roulement, les plus vieux seraient supprimés. La logique de suppression détermine les fichiers à supprimer selon l'intervalle, donc changer cette valeur peut conduire à ce que certains fichiers anciens restent sur le disque.

Si *delay* est à *true*, alors l'ouverture du fichier est reportée au premier appel de `emit()`.

Si le roulement doit se produire « à minuit » ou « à un jour fixe de la semaine » et si *atTime* n'est pas `None`, ce doit être une instance de `datetime.time` qui définit l'heure de la journée à laquelle le roulement se produit. Dans ce cas, la valeur de *atTime* ne sert qu'à déterminer la date du roulement *initial*, la date des roulements ultérieurs est déterminée par le calcul standard de l'intervalle.

Si *errors* est définit, il définit comment traiter les erreurs d'encodage.

> **Note**
>
> la date du premier roulement est déterminée lors de l'initialisation du gestionnaire. Ce n'est que lors d'un roulement que la date du roulement suivant n'est déterminée, et un roulement n'a lieu que si des entrées doivent être journalisées. Gardez bien cela à l'esprit pour ne pas avoir de surprises. Par exemple, un gestionnaire avec un intervalle « d'une minute » ne produira pas nécessairement des fichiers avec des dates (dans le nom desdits fichiers) séparées d'une minute. Si une application génère des journaux plus fréquemment que toutes les minutes au cours de son exécution, alors *dans ce cas* vous obtiendrez des fichiers avec des temps séparés d'une minute. Si la même application ne produit une entrée de journal que toutes les cinq minutes, il y aura des sauts dans les dates des fichiers produits qui correspondront aux moments où rien n'a été produit (et donc où aucun roulement n'a eu lieu).

Modifié dans la version 3.4 : ajout du paramètre *atTime*.

Modifié dans la version 3.6 : L'argument *filename* accepte les objets `Path` aussi bien que les chaînes de caractères.

Modifié dans la version 3.9 : ajout du paramètre *errors*.

doRollover()

Effectue un roulement, comme décrit ci-dessus.

emit(*record*)

Écrit l'enregistrement dans le fichier, effectuant un roulement au besoin comme décrit précédemment.

getFilesToDelete()

Returns a list of filenames which should be deleted as part of rollover. These are the absolute paths of the oldest backup log files written by the handler.

16.6.8 Gestionnaire à connecteur — *SocketHandler*

The `SocketHandler` class, located in the `logging.handlers` module, sends logging output to a network socket. The base class uses a TCP socket.

class logging.handlers.**SocketHandler**(*host*, *port*)

Returns a new instance of the `SocketHandler` class intended to communicate with a remote machine whose address is given by *host* and *port*.

Modifié dans la version 3.4 : If `port` is specified as `None`, a Unix domain socket is created using the value in `host` - otherwise, a TCP socket is created.

close()

Closes the socket.

emit()

Pickles the record's attribute dictionary and writes it to the socket in binary format. If there is an error with the socket, silently drops the packet. If the connection was previously lost, re-establishes the connection. To unpickle the record at the receiving end into a `LogRecord`, use the `makeLogRecord()` function.

handleError()

Handles an error which has occurred during `emit()`. The most likely cause is a lost connection. Closes the socket so that we can retry on the next event.

makeSocket()

This is a factory method which allows subclasses to define the precise type of socket they want. The default implementation creates a TCP socket (`socket.SOCK_STREAM`).

makePickle(*record*)

Pickles the record's attribute dictionary in binary format with a length prefix, and returns it ready for transmission across the socket. The details of this operation are equivalent to :

```
data = pickle.dumps(record_attr_dict, 1)
datalen = struct.pack('>L', len(data))
return datalen + data
```

Note that pickles aren't completely secure. If you are concerned about security, you may want to override this method to implement a more secure mechanism. For example, you can sign pickles using HMAC and then verify them on the receiving end, or alternatively you can disable unpickling of global objects on the receiving end.

send(*packet*)

Send a pickled byte-string *packet* to the socket. The format of the sent byte-string is as described in the documentation for `makePickle()`.

This function allows for partial sends, which can happen when the network is busy.

createSocket()

Tries to create a socket ; on failure, uses an exponential back-off algorithm. On initial failure, the handler will drop the message it was trying to send. When subsequent messages are handled by the same instance, it will not try connecting until some time has passed. The default parameters are such that the initial delay is one second, and if after that delay the connection still can't be made, the handler will double the delay each time up to a maximum of 30 seconds.

This behaviour is controlled by the following handler attributes :
— `retryStart` (initial delay, defaulting to 1.0 seconds).
— `retryFactor` (multiplier, defaulting to 2.0).
— `retryMax` (maximum delay, defaulting to 30.0 seconds).
This means that if the remote listener starts up *after* the handler has been used, you could lose messages (since the handler won't even attempt a connection until the delay has elapsed, but just silently drop messages during the delay period).

16.6.9 DatagramHandler

The `DatagramHandler` class, located in the `logging.handlers` module, inherits from `SocketHandler` to support sending logging messages over UDP sockets.

class logging.handlers.**DatagramHandler**(*host*, *port*)

Returns a new instance of the `DatagramHandler` class intended to communicate with a remote machine whose address is given by *host* and *port*.

> **Note**
>
> As UDP is not a streaming protocol, there is no persistent connection between an instance of this handler and *host*. For this reason, when using a network socket, a DNS lookup might have to be made each time an event is logged, which can introduce some latency into the system. If this affects you, you can do a lookup yourself and initialize this handler using the looked-up IP address rather than the hostname.

Modifié dans la version 3.4 : If `port` is specified as `None`, a Unix domain socket is created using the value in `host` - otherwise, a UDP socket is created.

emit()

Pickles the record's attribute dictionary and writes it to the socket in binary format. If there is an error with the socket, silently drops the packet. To unpickle the record at the receiving end into a `LogRecord`, use the `makeLogRecord()` function.

makeSocket()

The factory method of `SocketHandler` is here overridden to create a UDP socket (`socket.SOCK_DGRAM`).

send(*s*)

Send a pickled byte-string to a socket. The format of the sent byte-string is as described in the documentation for `SocketHandler.makePickle()`.

16.6.10 SysLogHandler

The `SysLogHandler` class, located in the `logging.handlers` module, supports sending logging messages to a remote or local Unix syslog.

class logging.handlers.**SysLogHandler**(*address=('localhost', SYSLOG_UDP_PORT)*, *facility=LOG_USER*, *socktype=socket.SOCK_DGRAM*, *timeout=None*)

Returns a new instance of the `SysLogHandler` class intended to communicate with a remote Unix machine whose address is given by *address* in the form of a `(host, port)` tuple. If *address* is not specified, `('localhost', 514)` is used. The address is used to open a socket. An alternative to providing a `(host, port)` tuple is providing an address as a string, for example '/dev/log'. In this case, a Unix domain socket is used to send the message to the syslog. If *facility* is not specified, `LOG_USER` is used. The type of socket opened depends on the *socktype* argument, which defaults to `socket.SOCK_DGRAM` and thus opens a UDP socket. To open a TCP socket (for use with the newer syslog daemons such as rsyslog), specify a value of `socket.SOCK_STREAM`. If *timeout* is specified, it sets a timeout (in seconds) for the socket operations. This can help prevent the program from hanging indefinitely if the syslog server is unreachable. By default, *timeout* is `None`, meaning no timeout is applied.

Note that if your server is not listening on UDP port 514, `SysLogHandler` may appear not to work. In that case, check what address you should be using for a domain socket - it's system dependent. For example, on Linux it's usually '/dev/log' but on OS/X it's '/var/run/syslog'. You'll need to check your platform and use the appropriate address (you may need to do this check at runtime if your application needs to run on several platforms). On Windows, you pretty much have to use the UDP option.

> **Note**
>
> On macOS 12.x (Monterey), Apple has changed the behaviour of their syslog daemon - it no longer listens on a domain socket. Therefore, you cannot expect `SysLogHandler` to work on this system.
>
> See gh-91070 for more information.

Modifié dans la version 3.2 : *socktype* was added.

Modifié dans la version 3.14 : *timeout* was added.

close()

Closes the socket to the remote host.

createSocket()

Tries to create a socket and, if it's not a datagram socket, connect it to the other end. This method is called during handler initialization, but it's not regarded as an error if the other end isn't listening at this point - the method will be called again when emitting an event, if there is no socket at that point.

Ajouté dans la version 3.11.

emit(*record*)

The record is formatted, and then sent to the syslog server. If exception information is present, it is *not*

sent to the server.

Modifié dans la version 3.2.1 : (See : bpo-12168.) In earlier versions, the message sent to the syslog daemons was always terminated with a NUL byte, because early versions of these daemons expected a NUL terminated message - even though it's not in the relevant specification (**RFC 5424**). More recent versions of these daemons don't expect the NUL byte but strip it off if it's there, and even more recent daemons (which adhere more closely to RFC 5424) pass the NUL byte on as part of the message.

To enable easier handling of syslog messages in the face of all these differing daemon behaviours, the appending of the NUL byte has been made configurable, through the use of a class-level attribute, `append_nul`. This defaults to `True` (preserving the existing behaviour) but can be set to `False` on a `SysLogHandler` instance in order for that instance to *not* append the NUL terminator.

Modifié dans la version 3.3 : (See : bpo-12419.) In earlier versions, there was no facility for an "ident" or "tag" prefix to identify the source of the message. This can now be specified using a class-level attribute, defaulting to `""` to preserve existing behaviour, but which can be overridden on a `SysLogHandler` instance in order for that instance to prepend the ident to every message handled. Note that the provided ident must be text, not bytes, and is prepended to the message exactly as is.

encodePriority(*facility*, *priority*)

Encodes the facility and priority into an integer. You can pass in strings or integers - if strings are passed, internal mapping dictionaries are used to convert them to integers.

The symbolic `LOG_` values are defined in *`SysLogHandler`* and mirror the values defined in the `sys/syslog.h` header file.

Priorities

Name (string)	Symbolic value
`alert`	LOG_ALERT
`crit` ou `critical`	LOG_CRIT
`debug`	LOG_DEBUG
`emerg` ou `panic`	LOG_EMERG
`err` ou `error`	LOG_ERR
`info`	LOG_INFO
`notice`	LOG_NOTICE
`warn` ou `warning`	LOG_WARNING

Facilities

Name (string)	Symbolic value
auth	LOG_AUTH
authpriv	LOG_AUTHPRIV
cron	LOG_CRON
daemon	LOG_DAEMON
ftp	LOG_FTP
kern	LOG_KERN
lpr	LOG_LPR
mail	LOG_MAIL
news	LOG_NEWS
syslog	LOG_SYSLOG
user	LOG_USER
uucp	LOG_UUCP
local0	LOG_LOCAL0
local1	LOG_LOCAL1
local2	LOG_LOCAL2
local3	LOG_LOCAL3
local4	LOG_LOCAL4
local5	LOG_LOCAL5
local6	LOG_LOCAL6
local7	LOG_LOCAL7

mapPriority(*levelname*)

Maps a logging level name to a syslog priority name. You may need to override this if you are using custom levels, or if the default algorithm is not suitable for your needs. The default algorithm maps `DEBUG`, `INFO`, `WARNING`, `ERROR` and `CRITICAL` to the equivalent syslog names, and all other level names to 'warning'.

16.6.11 NTEventLogHandler

The `NTEventLogHandler` class, located in the `logging.handlers` module, supports sending logging messages to a local Windows NT, Windows 2000 or Windows XP event log. Before you can use it, you need Mark Hammond's Win32 extensions for Python installed.

class logging.handlers.**NTEventLogHandler**(*appname*, *dllname=None*, *logtype='Application'*)

Returns a new instance of the `NTEventLogHandler` class. The *appname* is used to define the application name as it appears in the event log. An appropriate registry entry is created using this name. The *dllname* should give the fully qualified pathname of a .dll or .exe which contains message definitions to hold in the log (if not specified, `'win32service.pyd'` is used - this is installed with the Win32 extensions and contains some basic placeholder message definitions. Note that use of these placeholders will make your event logs big, as the entire message source is held in the log. If you want slimmer logs, you have to pass in the name of your own .dll or .exe which contains the message definitions you want to use in the event log). The *logtype* is one of `'Application'`, `'System'` or `'Security'`, and defaults to `'Application'`.

close()

At this point, you can remove the application name from the registry as a source of event log entries. However, if you do this, you will not be able to see the events as you intended in the Event Log Viewer - it needs to be able to access the registry to get the .dll name. The current version does not do this.

emit(*record*)

Determines the message ID, event category and event type, and then logs the message in the NT event log.

getEventCategory(*record*)

Returns the event category for the record. Override this if you want to specify your own categories. This version returns 0.

getEventType(*record*)

Returns the event type for the record. Override this if you want to specify your own types. This version

does a mapping using the handler's typemap attribute, which is set up in `__init__()` to a dictionary which contains mappings for `DEBUG`, `INFO`, `WARNING`, `ERROR` and `CRITICAL`. If you are using your own levels, you will either need to override this method or place a suitable dictionary in the handler's *typemap* attribute.

getMessageID(*record*)

Returns the message ID for the record. If you are using your own messages, you could do this by having the *msg* passed to the logger being an ID rather than a format string. Then, in here, you could use a dictionary lookup to get the message ID. This version returns 1, which is the base message ID in `win32service.pyd`.

16.6.12 SMTPHandler

The `SMTPHandler` class, located in the `logging.handlers` module, supports sending logging messages to an email address via SMTP.

class logging.handlers.**SMTPHandler**(*mailhost*, *fromaddr*, *toaddrs*, *subject*, *credentials=None*, *secure=None*, *timeout=1.0*)

Returns a new instance of the `SMTPHandler` class. The instance is initialized with the from and to addresses and subject line of the email. The *toaddrs* should be a list of strings. To specify a non-standard SMTP port, use the (host, port) tuple format for the *mailhost* argument. If you use a string, the standard SMTP port is used. If your SMTP server requires authentication, you can specify a (username, password) tuple for the *credentials* argument.

To specify the use of a secure protocol (TLS), pass in a tuple to the *secure* argument. This will only be used when authentication credentials are supplied. The tuple should be either an empty tuple, or a single-value tuple with the name of a keyfile, or a 2-value tuple with the names of the keyfile and certificate file. (This tuple is passed to the `smtplib.SMTP.starttls()` method.)

A timeout can be specified for communication with the SMTP server using the *timeout* argument.

Modifié dans la version 3.3 : Added the *timeout* parameter.

emit(*record*)

Formats the record and sends it to the specified addressees.

getSubject(*record*)

If you want to specify a subject line which is record-dependent, override this method.

16.6.13 MemoryHandler

The `MemoryHandler` class, located in the `logging.handlers` module, supports buffering of logging records in memory, periodically flushing them to a *target* handler. Flushing occurs whenever the buffer is full, or when an event of a certain severity or greater is seen.

`MemoryHandler` is a subclass of the more general `BufferingHandler`, which is an abstract class. This buffers logging records in memory. Whenever each record is added to the buffer, a check is made by calling `shouldFlush()` to see if the buffer should be flushed. If it should, then `flush()` is expected to do the flushing.

class logging.handlers.**BufferingHandler**(*capacity*)

Initializes the handler with a buffer of the specified capacity. Here, *capacity* means the number of logging records buffered.

emit(*record*)

Append the record to the buffer. If `shouldFlush()` returns true, call `flush()` to process the buffer.

flush()

For a `BufferingHandler` instance, flushing means that it sets the buffer to an empty list. This method can be overwritten to implement more useful flushing behavior.

shouldFlush(*record*)

Return `True` if the buffer is up to capacity. This method can be overridden to implement custom flushing strategies.

class logging.handlers.**MemoryHandler**(*capacity*, *flushLevel=ERROR*, *target=None*, *flushOnClose=True*)

Returns a new instance of the `MemoryHandler` class. The instance is initialized with a buffer size of *capacity* (number of records buffered). If *flushLevel* is not specified, `ERROR` is used. If no *target* is specified, the target will need to be set using `setTarget()` before this handler does anything useful. If *flushOnClose* is specified as `False`, then the buffer is *not* flushed when the handler is closed. If not specified or specified as `True`, the previous behaviour of flushing the buffer will occur when the handler is closed.

Modifié dans la version 3.6 : The *flushOnClose* parameter was added.

close()

Calls `flush()`, sets the target to `None` and clears the buffer.

flush()

For a `MemoryHandler` instance, flushing means just sending the buffered records to the target, if there is one. The buffer is also cleared when buffered records are sent to the target. Override if you want different behavior.

setTarget(*target*)

Sets the target handler for this handler.

shouldFlush(*record*)

Checks for buffer full or a record at the *flushLevel* or higher.

16.6.14 HTTPHandler

The `HTTPHandler` class, located in the `logging.handlers` module, supports sending logging messages to a web server, using either `GET` or `POST` semantics.

class logging.handlers.**HTTPHandler**(*host*, *url*, *method='GET'*, *secure=False*, *credentials=None*, *context=None*)

Returns a new instance of the `HTTPHandler` class. The *host* can be of the form `host:port`, should you need to use a specific port number. If no *method* is specified, `GET` is used. If *secure* is true, a HTTPS connection will be used. The *context* parameter may be set to a `ssl.SSLContext` instance to configure the SSL settings used for the HTTPS connection. If *credentials* is specified, it should be a 2-tuple consisting of userid and password, which will be placed in a HTTP 'Authorization' header using Basic authentication. If you specify credentials, you should also specify secure=True so that your userid and password are not passed in cleartext across the wire.

Modifié dans la version 3.5 : The *context* parameter was added.

mapLogRecord(*record*)

Provides a dictionary, based on `record`, which is to be URL-encoded and sent to the web server. The default implementation just returns `record.__dict__`. This method can be overridden if e.g. only a subset of `LogRecord` is to be sent to the web server, or if more specific customization of what's sent to the server is required.

emit(*record*)

Sends the record to the web server as a URL-encoded dictionary. The `mapLogRecord()` method is used to convert the record to the dictionary to be sent.

> **Note**
>
> Since preparing a record for sending it to a web server is not the same as a generic formatting operation, using `setFormatter()` to specify a `Formatter` for a `HTTPHandler` has no effect. Instead of calling `format()`, this handler calls `mapLogRecord()` and then `urllib.parse.urlencode()` to encode the dictionary in a form suitable for sending to a web server.

16.6.15 QueueHandler

Ajouté dans la version 3.2.

The `QueueHandler` class, located in the `logging.handlers` module, supports sending logging messages to a queue, such as those implemented in the `queue` or `multiprocessing` modules.

Along with the `QueueListener` class, `QueueHandler` can be used to let handlers do their work on a separate thread from the one which does the logging. This is important in web applications and also other service applications where threads servicing clients need to respond as quickly as possible, while any potentially slow operations (such as sending an email via `SMTPHandler`) are done on a separate thread.

class `logging.handlers.`**`QueueHandler`**(*queue*)

Returns a new instance of the `QueueHandler` class. The instance is initialized with the queue to send messages to. The *queue* can be any queue-like object ; it's used as-is by the `enqueue()` method, which needs to know how to send messages to it. The queue is not *required* to have the task tracking API, which means that you can use `SimpleQueue` instances for *queue*.

> **Note**
>
> If you are using `multiprocessing`, you should avoid using `SimpleQueue` and instead use `multiprocessing.Queue`.

emit(*record*)

Enqueues the result of preparing the LogRecord. Should an exception occur (e.g. because a bounded queue has filled up), the `handleError()` method is called to handle the error. This can result in the record silently being dropped (if `logging.raiseExceptions` is `False`) or a message printed to `sys.stderr` (if `logging.raiseExceptions` is `True`).

prepare(*record*)

Prepares a record for queuing. The object returned by this method is enqueued.

The base implementation formats the record to merge the message, arguments, exception and stack information, if present. It also removes unpickleable items from the record in-place. Specifically, it overwrites the record's `msg` and `message` attributes with the merged message (obtained by calling the handler's `format()` method), and sets the `args`, `exc_info` and `exc_text` attributes to `None`.

You might want to override this method if you want to convert the record to a dict or JSON string, or send a modified copy of the record while leaving the original intact.

> **Note**
>
> The base implementation formats the message with arguments, sets the `message` and `msg` attributes to the formatted message and sets the `args` and `exc_text` attributes to `None` to allow pickling and to prevent further attempts at formatting. This means that a handler on the `QueueListener` side won't have the information to do custom formatting, e.g. of exceptions. You may wish to subclass `QueueHandler` and override this method to e.g. avoid setting `exc_text` to `None`. Note that the `message` / `msg` / `args` changes are related to ensuring the record is pickleable, and you might or might not be able to avoid doing that depending on whether your `args` are pickleable. (Note that you may have to consider not only your own code but also code in any libraries that you use.)

enqueue(*record*)

Enqueues the record on the queue using `put_nowait()` ; you may want to override this if you want to use blocking behaviour, or a timeout, or a customized queue implementation.

listener

When created via configuration using `dictConfig()`, this attribute will contain a `QueueListener` instance for use with this handler. Otherwise, it will be `None`.

Ajouté dans la version 3.12.

16.6.16 QueueListener

Ajouté dans la version 3.2.

The `QueueListener` class, located in the `logging.handlers` module, supports receiving logging messages from a queue, such as those implemented in the `queue` or `multiprocessing` modules. The messages are received from a queue in an internal thread and passed, on the same thread, to one or more handlers for processing. While `QueueListener` is not itself a handler, it is documented here because it works hand-in-hand with `QueueHandler`.

Along with the `QueueHandler` class, `QueueListener` can be used to let handlers do their work on a separate thread from the one which does the logging. This is important in web applications and also other service applications where threads servicing clients need to respond as quickly as possible, while any potentially slow operations (such as sending an email via `SMTPHandler`) are done on a separate thread.

class `logging.handlers.`**`QueueListener`**(*queue*, **handlers*, *respect_handler_level=False*)

Returns a new instance of the `QueueListener` class. The instance is initialized with the queue to send messages to and a list of handlers which will handle entries placed on the queue. The queue can be any queue-like object; it's passed as-is to the `dequeue()` method, which needs to know how to get messages from it. The queue is not *required* to have the task tracking API (though it's used if available), which means that you can use `SimpleQueue` instances for *queue*.

> **Note**
>
> If you are using `multiprocessing`, you should avoid using `SimpleQueue` and instead use `multiprocessing.Queue`.

If `respect_handler_level` is `True`, a handler's level is respected (compared with the level for the message) when deciding whether to pass messages to that handler; otherwise, the behaviour is as in previous Python versions - to always pass each message to each handler.

Modifié dans la version 3.5 : The `respect_handler_level` argument was added.

dequeue(*block*)

Dequeues a record and return it, optionally blocking.

The base implementation uses `get()`. You may want to override this method if you want to use timeouts or work with custom queue implementations.

prepare(*record*)

Prepare a record for handling.

This implementation just returns the passed-in record. You may want to override this method if you need to do any custom marshalling or manipulation of the record before passing it to the handlers.

handle(*record*)

Handle a record.

This just loops through the handlers offering them the record to handle. The actual object passed to the handlers is that which is returned from `prepare()`.

start()

Starts the listener.

This starts up a background thread to monitor the queue for LogRecords to process.

stop()

Stops the listener.

This asks the thread to terminate, and then waits for it to do so. Note that if you don't call this before your application exits, there may be some records still left on the queue, which won't be processed.

enqueue_sentinel()

Writes a sentinel to the queue to tell the listener to quit. This implementation uses `put_nowait()`. You

may want to override this method if you want to use timeouts or work with custom queue implementations.

Ajouté dans la version 3.3.

Voir aussi

Module *logging*
Référence d'API pour le module de journalisation.

Module *logging.config*
API de configuration pour le module de journalisation.

16.7 platform --- Access to underlying platform's identifying data

Code source : Lib/platform.py

Note

Les spécificités des plateformes sont regroupées dans des sections triées par ordre alphabétique. Linux est inclus dans la section Unix.

16.7.1 Multiplateformes

platform.**architecture**(*executable=sys.executable*, *bits=''*, *linkage=''*)

Interroge l'exécutable fourni (par défaut l'interpréteur Python) sur les informations de l'architecture.

Renvoie un *n*-uplet de `(bits, linkage)` qui contient de l'information sur l'architecture binaire et le format de lien. Les deux valeurs sont des chaînes de caractères.

Lorsqu'une valeur ne peut être déterminée, la valeur passée en paramètre est utilisée. Si la valeur passée à *bits* est `''`, la valeur de `sizeof(pointer)` (ou `sizeof(long)` sur les versions Python antérieures à 1.5.2) est utilisée comme indicateur de la taille de pointeur prise en charge.

La fonction dépend de la commande `file` du système pour accomplir la tâche. `file` est disponible sur quasiment toutes les plateformes Unix ainsi que sur certaines plateformes hors de la famille Unix et l'exécutable doit pointer vers l'interpréteur Python. Des valeurs par défaut raisonnables sont utilisées lorsque les conditions précédentes ne sont pas atteintes.

Note

On macOS (and perhaps other platforms), executable files may be universal files containing multiple architectures.

To get at the "64-bitness" of the current interpreter, it is more reliable to query the *sys.maxsize* attribute :

```
is_64bits = sys.maxsize > 2**32
```

platform.**machine**()

Returns the machine type, e.g. `'AMD64'`. An empty string is returned if the value cannot be determined.

platform.**node**()

Renvoie le nom de l'ordinateur sur le réseau (pas forcément pleinement qualifié). Une chaîne de caractères vide est renvoyée s'il ne peut pas être déterminé.

platform.**platform**(*aliased=False*, *terse=False*)

Renvoie une chaîne de caractère identifiant la plateforme avec le plus d'informations possible.

La valeur renvoyée est destinée à la *lecture humaine* plutôt que l'interprétation machine. Il est possible qu'elle soit différente selon la plateforme et c'est voulu.

Si *aliased* est vrai, la fonction utilisera des alias pour certaines plateformes qui utilisent des noms de système qui diffèrent de leurs noms communs. Par exemple, SunOS sera reconnu comme Solaris. La fonction `system_alias()` est utilisée pour l'implémentation.

Si *terse* est vrai, la fonction ne renverra que l'information nécessaire à l'identification de la plateforme.

Modifié dans la version 3.8 : Sur MacOS, la fonction essaie maintenant d'utiliser la fonction `mac_ver()` pour obtenir la version de MacOS plutôt que la version de Darwin : le résultat de `mac_ver` est utilisé si ce n'est pas une chaîne vide.

platform.**processor**()

Renvoie le (vrai) nom du processeur. Par exemple : `'amdk6'`.

Une chaîne de caractères vide est renvoyée si la valeur ne peut être déterminée. Prenez note que plusieurs plateformes ne fournissent pas cette information ou renvoient la même valeur que la fonction `machine()`. NetBSD agit ainsi.

platform.**python_build**()

Renvoie une paire `(buildno, builddate)` de chaîne de caractères identifiant la version et la date de compilation de Python.

platform.**python_compiler**()

Renvoie une chaîne de caractères identifiant le compilateur utilisé pour compiler Python.

platform.**python_branch**()

Renvoie la chaîne de caractères identifiant la branche du gestionnaire de versions de l'implémentation Python.

platform.**python_implementation**()

Renvoie une chaîne de caractères identifiant l'implémentation de Python. Des valeurs possibles sont : `CPython`, `IronPython`, `Jython`, `Pypy`.

platform.**python_revision**()

Renvoie la chaîne de caractères identifiant la révision du gestionnaire de versions de l'implémentation Python.

platform.**python_version**()

Renvoie la version de Python comme une chaîne de caractères `'major.minor.patchlevel'`.

Prenez note que la valeur renvoyée inclut toujours le *patchlevel* (valeur par défaut de 0) à la différence de `sys.version`.

platform.**python_version_tuple**()

Renvoie la version de Python comme un triplet de chaînes de caractères `(major, minor, patchlevel)`.

Prenez note que la valeur renvoyée inclut toujours le *patchlevel* (valeur par défaut de `'0'`) à la différence de `sys.version`.

platform.**release**()

Returns the system's release, e.g. `'2.2.0'` or `'NT'`. An empty string is returned if the value cannot be determined.

platform.**system**()

Renvoie le nom du système d'exploitation, tel que `'Linux'`, `'Darwin'`, `'Java'`, `'Windows'`. Une chaîne de caractères vide est renvoyée si le nom ne peut être déterminé.

On iOS and Android, this returns the user-facing OS name (i.e, `'iOS'`, `'iPadOS'` or `'Android'`). To obtain the kernel name (`'Darwin'` or `'Linux'`), use `os.uname()`.

platform.**system_alias**(*system*, *release*, *version*)

Renvoie `(system, release, version)` avec des alias pour les noms communs de certains systèmes. Modifie aussi l'ordre de l'information pour éviter la confusion.

platform.**version**()

Renvoie la version de déploiement du système. Par exemple, `'#3 on degas'`. Une chaîne de caractères vide est renvoyée si aucune valeur ne peut être déterminée.

On iOS and Android, this is the user-facing OS version. To obtain the Darwin or Linux kernel version, use `os.uname()`.

platform.**uname**()

Interface de *uname* relativement portable. Renvoie un `namedtuple()` contenant six attributs : `system`, `node`, `release`, `version`, `machine` et `processor`.

`processor` is resolved late, on demand.

Note : the first two attribute names differ from the names presented by `os.uname()`, where they are named `sysname` and `nodename`.

Les entrées qui ne peuvent pas être identifiées ont la valeur `''`.

Modifié dans la version 3.3 : Result changed from a tuple to a `namedtuple()`.

Modifié dans la version 3.9 : `processor` is resolved late instead of immediately.

16.7.2 Plateforme Java

platform.**java_ver**(*release=''*, *vendor=''*, *vminfo=('', '', '')*, *osinfo=('', '', '')*)

Version de l'interface pour Jython.

Renvoie un *n*-uplet `(release, vendor, vminfo, osinfo)`. *vminfo* est un triplet de valeur `(vm_name, vm_release, vm_vendor)` et *osinfo* est un triplet de valeur `(os_name, os_version, os_arch)`. Les valeurs indéterminables ont la valeur des paramètres par défaut (valeur de `''` par défaut).

Deprecated since version 3.13, will be removed in version 3.15 : It was largely untested, had a confusing API, and was only useful for Jython support.

16.7.3 Plateforme Windows

platform.**win32_ver**(*release=''*, *version=''*, *csd=''*, *ptype=''*)

Get additional version information from the Windows Registry and return a tuple `(release, version, csd, ptype)` referring to OS release, version number, CSD level (service pack) and OS type (multi/single processor). Values which cannot be determined are set to the defaults given as parameters (which all default to an empty string).

As a hint : *ptype* is `'Uniprocessor Free'` on single processor NT machines and `'Multiprocessor Free'` on multi processor machines. The `'Free'` refers to the OS version being free of debugging code. It could also state `'Checked'` which means the OS version uses debugging code, i.e. code that checks arguments, ranges, etc.

platform.**win32_edition**()

Returns a string representing the current Windows edition, or `None` if the value cannot be determined. Possible values include but are not limited to `'Enterprise'`, `'IoTUAP'`, `'ServerStandard'`, and `'nanoserver'`.

Ajouté dans la version 3.8.

platform.**win32_is_iot**()

Renvoie `True` si l'édition de Windows renvoyée par la fonction `win32_edition()` est reconnue comme une édition IoT.

Ajouté dans la version 3.8.

16.7.4 macOS Platform

platform.**mac_ver**(*release=''*, *versioninfo=('', '', '')*, *machine=''*)

Get macOS version information and return it as tuple `(release, versioninfo, machine)` with *versioninfo* being a tuple `(version, dev_stage, non_release_version)`.

Les entrées qui ne peuvent pas être identifiées auront la valeur `''`. Les membres du *n*-uplet sont tous des chaînes de caractères.

16.7.5 iOS Platform

platform.**ios_ver**(*system=''*, *release=''*, *model=''*, *is_simulator=False*)

Get iOS version information and return it as a `namedtuple()` with the following attributes :

— `system` is the OS name ; either `'iOS'` or `'iPadOS'`.
— `release` is the iOS version number as a string (e.g., `'17.2'`).
— `model` is the device model identifier ; this will be a string like `'iPhone13,2'` for a physical device, or `'iPhone'` on a simulator.
— `is_simulator` is a boolean describing if the app is running on a simulator or a physical device.

Entries which cannot be determined are set to the defaults given as parameters.

16.7.6 Plateformes Unix

platform.**libc_ver**(*executable=sys.executable*, *lib=''*, *version=''*, *chunksize=16384*)

Tente d'identifier la version de la bibliothèque standard C à laquelle le fichier exécutable (par défaut l'interpréteur Python) est lié. Renvoie une paire de chaînes de caractères `(lib, version)`. Les valeurs passées en paramètre seront retournées si la recherche échoue.

Prenez note que cette fonction a une connaissance profonde des méthodes utilisées par les versions de la bibliothèque standard C pour ajouter des symboles au fichier exécutable. Elle n'est probablement utilisable qu'avec des exécutables compilés avec **gcc**.

Le fichier est lu en blocs de *chunksize* octets.

16.7.7 Linux Platforms

platform.**freedesktop_os_release**()

Get operating system identification from `os-release` file and return it as a dict. The `os-release` file is a freedesktop.org standard and is available in most Linux distributions. A noticeable exception is Android and Android-based distributions.

Raises `OSError` or subclass when neither `/etc/os-release` nor `/usr/lib/os-release` can be read.

On success, the function returns a dictionary where keys and values are strings. Values have their special characters like `"` and `$` unquoted. The fields `NAME`, `ID`, and `PRETTY_NAME` are always defined according to the standard. All other fields are optional. Vendors may include additional fields.

Note that fields like `NAME`, `VERSION`, and `VARIANT` are strings suitable for presentation to users. Programs should use fields like `ID`, `ID_LIKE`, `VERSION_ID`, or `VARIANT_ID` to identify Linux distributions.

Exemple :

```
def get_like_distro():
    info = platform.freedesktop_os_release()
    ids = [info["ID"]]
    if "ID_LIKE" in info:
        # ids are space separated and ordered by precedence
        ids.extend(info["ID_LIKE"].split())
    return ids
```

Ajouté dans la version 3.10.

16.7.8 Android Platform

platform.**android_ver**(*release=''*, *api_level=0*, *manufacturer=''*, *model=''*, *device=''*, *is_emulator=False*)

Get Android device information. Returns a `namedtuple()` with the following attributes. Values which cannot be determined are set to the defaults given as parameters.

— `release` - Android version, as a string (e.g. `"14"`).
— `api_level` - API level of the running device, as an integer (e.g. `34` for Android 14). To get the API level which Python was built against, see `sys.getandroidapilevel()`.
— `manufacturer` - Manufacturer name.
— `model` - Model name – typically the marketing name or model number.
— `device` - Device name – typically the model number or a codename.
— `is_emulator` - `True` if the device is an emulator ; `False` if it's a physical device.

Google maintains a list of known model and device names.

Ajouté dans la version 3.13.

16.7.9 Miscellaneous

platform.**invalidate_caches**()

Clear out the internal cache of information, such as the `uname()`. This is typically useful when the platform's `node()` is changed by an external process and one needs to retrieve the updated value.

Ajouté dans la version 3.14.

16.8 `errno` --- Standard errno system symbols

This module makes available standard `errno` system symbols. The value of each symbol is the corresponding integer value. The names and descriptions are borrowed from `linux/include/errno.h`, which should be all-inclusive.

errno.**errorcode**

Dictionnaire associant la valeur *errno* au nom de chaîne dans le système sous-jacent. Par exemple, `errno.errorcode[errno.EPERM]` correspond à `'EPERM'`.

Pour traduire un code d'erreur en message d'erreur, utilisez `os.strerror()`.

De la liste suivante, les symboles qui ne sont pas utilisés dans la plateforme actuelle ne sont pas définis par le module. La liste spécifique des symboles définis est disponible comme `errno.errorcode.keys()`. Les symboles disponibles font partie de cette liste :

errno.**EPERM**

Operation not permitted. This error is mapped to the exception `PermissionError`.

errno.**ENOENT**

No such file or directory. This error is mapped to the exception `FileNotFoundError`.

errno.**ESRCH**

No such process. This error is mapped to the exception `ProcessLookupError`.

errno.**EINTR**

Interrupted system call. This error is mapped to the exception `InterruptedError`.

errno.**EIO**

Erreur d'entrée-sortie

errno.**ENXIO**

Dispositif ou adresse inexistant

errno.**E2BIG**

Liste d'arguments trop longue

errno.**ENOEXEC**

Erreur de format d'exécution

errno.**EBADF**

Mauvais descripteur de fichier

errno.**ECHILD**

No child processes. This error is mapped to the exception *ChildProcessError*.

errno.**EAGAIN**

Try again. This error is mapped to the exception *BlockingIOError*.

errno.**ENOMEM**

Mémoire insuffisante

errno.**EACCES**

Permission denied. This error is mapped to the exception *PermissionError*.

errno.**EFAULT**

Mauvaise adresse

errno.**ENOTBLK**

Dispositif de bloc requis

errno.**EBUSY**

Dispositif ou ressource occupé

errno.**EEXIST**

File exists. This error is mapped to the exception *FileExistsError*.

errno.**EXDEV**

Lien inapproprié

errno.**ENODEV**

Dispositif inexistant

errno.**ENOTDIR**

Not a directory. This error is mapped to the exception *NotADirectoryError*.

errno.**EISDIR**

Is a directory. This error is mapped to the exception *IsADirectoryError*.

errno.**EINVAL**

Argument invalide

errno.**ENFILE**

Plus de descripteur de fichier disponible

errno.**EMFILE**

Trop de fichiers ouverts

errno.**ENOTTY**

Opération de contrôle d'entrée-sortie invalide

errno.**ETXTBSY**

Fichier texte occupé

errno.**EFBIG**

Fichier trop grand

errno.**ENOSPC**

Plus de place sur le dispositif

errno.**ESPIPE**
Recherche invalide

errno.**EROFS**
Système de fichiers en lecture seule

errno.**EMLINK**
Trop de liens symboliques

errno.**EPIPE**
Broken pipe. This error is mapped to the exception *BrokenPipeError*.

errno.**EDOM**
Argument mathématique hors du domaine de définition de la fonction

errno.**ERANGE**
Résultat mathématique non représentable

errno.**EDEADLK**
Un interblocage se produirait sur cette ressource

errno.**ENAMETOOLONG**
Nom de fichier trop long

errno.**ENOLCK**
Plus de verrou de fichier disponible

errno.**ENOSYS**
Fonction non implémentée

errno.**ENOTEMPTY**
Dossier non vide

errno.**ELOOP**
Trop de liens symboliques trouvés

errno.**EWOULDBLOCK**
Operation would block. This error is mapped to the exception *BlockingIOError*.

errno.**ENOMSG**
Pas de message du type voulu

errno.**EIDRM**
Identifiant supprimé

errno.**ECHRNG**
Le numéro de canal est hors des limites

errno.**EL2NSYNC**
Le niveau 2 n'est pas synchronisé

errno.**EL3HLT**
Niveau 3 stoppé

errno.**EL3RST**
Niveau 3 réinitialisé

errno.**ELNRNG**
Le numéro du lien est hors des limites

errno.**EUNATCH**
Le pilote de protocole n'est pas attaché

`errno.ENOCSI`
Pas de structure *CSI* disponible

`errno.EL2HLT`
Niveau 2 stoppé

`errno.EBADE`
Échange invalide

`errno.EBADR`
Descripteur de requête invalide

`errno.EXFULL`
Échange complet

`errno.ENOANO`
Pas de *anode*

`errno.EBADRQC`
Code de requête invalide

`errno.EBADSLT`
Slot invalide

`errno.EDEADLOCK`
Interblocage lors du verrouillage de fichier

`errno.EBFONT`
Mauvais format de fichier de police

`errno.ENOSTR`
Le périphérique n'est pas un flux

`errno.ENODATA`
Pas de donnée disponible

`errno.ETIME`
Délai maximal atteint

`errno.ENOSR`
Pas assez de ressources de type flux

`errno.ENONET`
Machine hors réseau

`errno.ENOPKG`
Paquet non installé

`errno.EREMOTE`
L'objet est distant

`errno.ENOLINK`
Lien coupé

`errno.EADV`
Erreur d'annonce

`errno.ESRMNT`
Erreur *Srmount*

`errno.ECOMM`
Erreur de communication lors de l'envoi

errno.**EPROTO**
Erreur de protocole

errno.**EMULTIHOP**
Transfert à sauts multiples essayé

errno.**EDOTDOT**
erreur spécifique *RFS*

errno.**EBADMSG**
Pas un message de données

errno.**EOVERFLOW**
Valeur trop grande pour être stockée dans ce type de donnée

errno.**ENOTUNIQ**
Nom non-unique dans le réseau

errno.**EBADFD**
Descripteur de fichier en mauvais état

errno.**EREMCHG**
Adresse distante changée

errno.**ELIBACC**
Accès impossible à une bibliothèque partagée nécessaire

errno.**ELIBBAD**
Accès à une bibliothèque partagée corrompue

errno.**ELIBSCN**
Section *.lib* de *a.out* corrompue

errno.**ELIBMAX**
Tentative de liaison entre trop de bibliothèques partagées

errno.**ELIBEXEC**
Impossible d'exécuter directement une bibliothèque partagée

errno.**EILSEQ**
Séquence de *bytes* illégale

errno.**ERESTART**
Appel système interrompu qui devrait être relancé

errno.**ESTRPIPE**
Erreur d'enchaînement de flux

errno.**EUSERS**
Trop d'utilisateurs

errno.**ENOTSOCK**
Opération d'interface de connexion alors que ce n'est pas une interface de connexion

errno.**EDESTADDRREQ**
Adresse de destination obligatoire

errno.**EMSGSIZE**
Message trop long

errno.**EPROTOTYPE**
Mauvais type de protocole pour ce connecteur

errno.**ENOPROTOOPT**
Protocole pas disponible

errno.**EPROTONOSUPPORT**
Protocole non géré

errno.**ESOCKTNOSUPPORT**
Type de connecteur non géré

errno.**EOPNOTSUPP**
Opération non gérée par cette fin de lien

errno.**ENOTSUP**
Operation not supported

Ajouté dans la version 3.2.

errno.**EPFNOSUPPORT**
Famille de protocole non gérée

errno.**EAFNOSUPPORT**
Famille d'adresses non gérée par ce protocole

errno.**EADDRINUSE**
Adresse déjà utilisée

errno.**EADDRNOTAVAIL**
Impossible d'assigner l'adresse demandée

errno.**ENETDOWN**
Le réseau est désactivé

errno.**ENETUNREACH**
Réseau inaccessible

errno.**ENETRESET**
Connexion annulée par le réseau

errno.**ECONNABORTED**
Software caused connection abort. This error is mapped to the exception `ConnectionAbortedError`.

errno.**ECONNRESET**
Connection reset by peer. This error is mapped to the exception `ConnectionResetError`.

errno.**ENOBUFS**
Plus d'espace tampon disponible

errno.**EISCONN**
L'interface de connexion est déjà connectée

errno.**ENOTCONN**
L'interface de connexion n'est pas connectée

errno.**ESHUTDOWN**
Cannot send after transport endpoint shutdown. This error is mapped to the exception `BrokenPipeError`.

errno.**ETOOMANYREFS**
Trop de descripteurs : impossible d'effectuer la liaison

errno.**ETIMEDOUT**
Connection timed out. This error is mapped to the exception `TimeoutError`.

errno.**ECONNREFUSED**

Connection refused. This error is mapped to the exception *ConnectionRefusedError*.

errno.**EHOSTDOWN**

Hôte éteint

errno.**EHOSTUNREACH**

Pas de route vers l'hôte

errno.**EHWPOISON**

Memory page has hardware error.

Ajouté dans la version 3.14.

errno.**EALREADY**

Operation already in progress. This error is mapped to the exception *BlockingIOError*.

errno.**EINPROGRESS**

Operation now in progress. This error is mapped to the exception *BlockingIOError*.

errno.**ESTALE**

Descripteur de fichier NFS corrompu

errno.**EUCLEAN**

La structure a besoin d'être nettoyée

errno.**ENOTNAM**

N'est pas un fichier nommé du type *XENIX*

errno.**ENAVAIL**

Pas de sémaphore *XENIX* disponible

errno.**EISNAM**

Est un fichier nommé

errno.**EREMOTEIO**

Erreur d'entrées-sorties distante

errno.**EDQUOT**

Quota dépassé

errno.**EQFULL**

Interface output queue is full

Ajouté dans la version 3.11.

errno.**ENOMEDIUM**

No medium found

errno.**EMEDIUMTYPE**

Wrong medium type

errno.**ENOKEY**

Required key not available

errno.**EKEYEXPIRED**

Key has expired

errno.**EKEYREVOKED**

Key has been revoked

errno.**EKEYREJECTED**

Key was rejected by service

errno.**ERFKILL**
Operation not possible due to RF-kill

errno.**ELOCKUNMAPPED**
Locked lock was unmapped

errno.**ENOTACTIVE**
Facility is not active

errno.**EAUTH**
Authentication error

Ajouté dans la version 3.2.

errno.**EBADARCH**
Bad CPU type in executable

Ajouté dans la version 3.2.

errno.**EBADEXEC**
Bad executable (or shared library)

Ajouté dans la version 3.2.

errno.**EBADMACHO**
Malformed Mach-o file

Ajouté dans la version 3.2.

errno.**EDEVERR**
Device error

Ajouté dans la version 3.2.

errno.**EFTYPE**
Inappropriate file type or format

Ajouté dans la version 3.2.

errno.**ENEEDAUTH**
Need authenticator

Ajouté dans la version 3.2.

errno.**ENOATTR**
Attribute not found

Ajouté dans la version 3.2.

errno.**ENOPOLICY**
Policy not found

Ajouté dans la version 3.2.

errno.**EPROCLIM**
Too many processes

Ajouté dans la version 3.2.

errno.**EPROCUNAVAIL**
Bad procedure for program

Ajouté dans la version 3.2.

errno.**EPROGMISMATCH**
Program version wrong

Ajouté dans la version 3.2.

errno.**EPROGUNAVAIL**

RPC prog. not avail

Ajouté dans la version 3.2.

errno.**EPWROFF**

Device power is off

Ajouté dans la version 3.2.

errno.**EBADRPC**

RPC struct is bad

Ajouté dans la version 3.2.

errno.**ERPCMISMATCH**

RPC version wrong

Ajouté dans la version 3.2.

errno.**ESHLIBVERS**

Shared library version mismatch

Ajouté dans la version 3.2.

errno.**ENOTCAPABLE**

Capabilities insufficient. This error is mapped to the exception `PermissionError`.

Availability : WASI, FreeBSD

Ajouté dans la version 3.11.1.

errno.**ECANCELED**

Operation canceled

Ajouté dans la version 3.2.

errno.**EOWNERDEAD**

Owner died

Ajouté dans la version 3.2.

errno.**ENOTRECOVERABLE**

State not recoverable

Ajouté dans la version 3.2.

16.9 `ctypes` --- A foreign function library for Python

Source code : Lib/ctypes

`ctypes` est une bibliothèque d'appel à des fonctions externes en python. Elle fournit des types de données compatibles avec le langage C et permet d'appeler des fonctions depuis des DLL ou des bibliothèques partagées, rendant ainsi possible l'interfaçage de ces bibliothèques avec du pur code Python.

16.9.1 Didacticiel de *ctypes*

Remarque : les exemples de code de ce didacticiel utilisent `doctest` pour s'assurer de leur propre bon fonctionnement. Vu que certains de ces exemples ont un comportement différent en Linux, Windows ou macOS, ils contiennent des directives *doctest* dans les commentaires.

Remarque : le type `c_int` du module apparaît dans certains de ces exemples. Sur les plates-formes où `sizeof(long) == sizeof(int)`, ce type est un alias de `c_long`. Ne soyez donc pas surpris si `c_long` s'affiche là où vous vous attendiez à `c_int` — il s'agit bien du même type.

Chargement des DLL

ctypes fournit l'objet *cdll* pour charger des bibliothèques à liens dynamiques (et les objets *windll* et *oledll* en Windows).

You load libraries by accessing them as attributes of these objects. *cdll* loads libraries which export functions using the standard `cdecl` calling convention, while *windll* libraries call functions using the `stdcall` calling convention. *oledll* also uses the `stdcall` calling convention, and assumes the functions return a Windows `HRESULT` error code. The error code is used to automatically raise an *OSError* exception when the function call fails.

Modifié dans la version 3.3 : En Windows, les erreurs levaient auparavant une *WindowsError*, qui est maintenant un alias de *OSError*.

Here are some examples for Windows. Note that `msvcrt` is the MS standard C library containing most standard C functions, and uses the `cdecl` calling convention :

```
>>> from ctypes import *
>>> print(windll.kernel32)
<WinDLL 'kernel32', handle ... at ...>
>>> print(cdll.msvcrt)
<CDLL 'msvcrt', handle ... at ...>
>>> libc = cdll.msvcrt
>>>
```

Windows ajoute le suffixe habituel `.dll` automatiquement.

> **Note**
>
> Accéder à la bibliothèque standard C par `cdll.msvcrt` utilise une version obsolète de la bibliothèque qui peut avoir des problèmes de compatibilité avec celle que Python utilise. Si possible, mieux vaut utiliser la fonctionnalité native de Python, ou bien importer et utiliser le module `msvcrt`.

On Linux, it is required to specify the filename *including* the extension to load a library, so attribute access can not be used to load libraries. Either the *LoadLibrary()* method of the dll loaders should be used, or you should load the library by creating an instance of CDLL by calling the constructor :

```
>>> cdll.LoadLibrary("libc.so.6")
<CDLL 'libc.so.6', handle ... at ...>
>>> libc = CDLL("libc.so.6")
>>> libc
<CDLL 'libc.so.6', handle ... at ...>
>>>
```

Accès aux fonctions des DLL chargées

Les fonctions sont alors des attributs des objets DLL :

```
>>> libc.printf
<_FuncPtr object at 0x...>
>>> print(windll.kernel32.GetModuleHandleA)
<_FuncPtr object at 0x...>
>>> print(windll.kernel32.MyOwnFunction)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "ctypes.py", line 239, in __getattr__
    func = _StdcallFuncPtr(name, self)
AttributeError: function 'MyOwnFunction' not found
>>>
```

Note that win32 system dlls like `kernel32` and `user32` often export ANSI as well as UNICODE versions of a function. The UNICODE version is exported with a `W` appended to the name, while the ANSI version is exported with an `A` appended to the name. The win32 `GetModuleHandle` function, which returns a *module handle* for a given module name, has the following C prototype, and a macro is used to expose one of them as `GetModuleHandle` depending on whether UNICODE is defined or not :

```
/* ANSI version */
HMODULE GetModuleHandleA(LPCSTR lpModuleName);
/* UNICODE version */
HMODULE GetModuleHandleW(LPCWSTR lpModuleName);
```

windll n'en choisit pas une par magie, il faut accéder à la bonne en écrivant explicitement `GetModuleHandleA` ou `GetModuleHandleW` et en les appelant ensuite avec des objets octets ou avec des chaînes de caractères, respectivement.

Les DLL exportent parfois des fonctions dont les noms ne sont pas des identifiants Python valides, comme `"??2@YAPAXI@Z"`. Dans ce cas, il faut utiliser *`getattr()`* pour accéder à la fonction :

```
>>> getattr(cdll.msvcrt, "??2@YAPAXI@Z")
<_FuncPtr object at 0x...>
>>>
```

Sous Windows, certaines DLL exportent des fonctions à travers un indice plutôt qu'à travers un nom. On accède à une fonction en indiçant l'objet DLL avec son index :

```
>>> cdll.kernel32[1]
<_FuncPtr object at 0x...>
>>> cdll.kernel32[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "ctypes.py", line 310, in __getitem__
    func = _StdcallFuncPtr(name, self)
AttributeError: function ordinal 0 not found
>>>
```

Appel de fonctions

You can call these functions like any other Python callable. This example uses the `rand()` function, which takes no arguments and returns a pseudo-random integer :

```
>>> print(libc.rand())
1804289383
```

On Windows, you can call the `GetModuleHandleA()` function, which returns a win32 module handle (passing `None` as single argument to call it with a `NULL` pointer) :

```
>>> print(hex(windll.kernel32.GetModuleHandleA(None)))
0x1d000000
>>>
```

Une *`ValueError`* est levée quand on appelle une fonction `stdcall` avec la convention d'appel `cdecl` et vice-versa :

```
>>> cdll.kernel32.GetModuleHandleA(None)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Procedure probably called with not enough arguments (4 bytes missing)
>>>

>>> windll.msvcrt.printf(b"spam")
```

(suite sur la page suivante)

(suite de la page précédente)

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Procedure probably called with too many arguments (4 bytes in excess)
>>>
```

Pour déterminer la convention d'appel, il faut consulter l'en-tête C ou la documentation de la fonction à appeler.

En Windows, `ctypes` tire profit de la gestion structurée des exceptions (*structured exception handling*) *win32* pour empêcher les plantages dus à des interruptions, afin de préserver la protection globale (*general protection faults*) du système, lorsque des fonctions sont appelées avec un nombre incorrect d'arguments :

```
>>> windll.kernel32.GetModuleHandleA(32)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OSError: exception: access violation reading 0x00000020
>>>
```

Cependant, il y a suffisamment de façons de faire planter Python avec `ctypes`, donc il faut être prudent dans tous les cas. Le module `faulthandler` est pratique pour déboguer les plantages (p. ex. dus à des erreurs de segmentation produites par des appels erronés à la bibliothèque C).

`None`, integers, bytes objects and (unicode) strings are the only native Python objects that can directly be used as parameters in these function calls. `None` is passed as a C `NULL` pointer, bytes objects and strings are passed as pointer to the memory block that contains their data (`char*` or `wchar_t*`). Python integers are passed as the platform's default C `int` type, their value is masked to fit into the C type.

Avant de poursuivre sur l'appel de fonctions avec d'autres types de paramètres, apprenons-en un peu plus sur les types de données de `ctypes`.

Types de données de base

`ctypes` définit plusieurs types de donnée de base compatibles avec le C :

Types de *ctypes*	Type C	Type Python
c_bool	_Bool	*bool* (1)
c_char	char	objet octet (*bytes*) de 1 caractère
c_wchar	wchar_t	chaîne de caractères (*string*) de longueur 1
c_byte	char	*int*
c_ubyte	unsigned char	*int*
c_short	short	*int*
c_ushort	unsigned short	*int*
c_int	int	*int*
c_uint	unsigned int	*int*
c_long	long	*int*
c_ulong	unsigned long	*int*
c_longlong	__int64 or long long	*int*
c_ulonglong	unsigned __int64 or unsigned long long	*int*
c_size_t	size_t	*int*
c_ssize_t	ssize_t or Py_ssize_t	*int*
c_time_t	time_t	*int*
c_float	float	*float*
c_double	double	*float*
c_longdouble	long double	*float*
c_char_p	char* (NUL terminated)	objet octet (*bytes*) ou None
c_wchar_p	wchar_t* (NUL terminated)	chaîne de caractères (*string*) ou None
c_void_p	void*	*int* ou None

(1) Le constructeur accepte n'importe quel objet convertible en booléen.

Additionally, if IEC 60559 compatible complex arithmetic (Annex G) is supported in both C and libffi, the following complex types are available :

Types de *ctypes*	Type C	Type Python
c_float_complex	float complex	complex
c_double_complex	double complex	complex
c_longdouble_complex	long double complex	complex

Il est possible de créer chacun de ces types en les appelant avec une valeur d'initialisation du bon type et avec une valeur cohérente :

```
>>> c_int()
c_long(0)
>>> c_wchar_p("Hello, World")
c_wchar_p(140018365411392)
>>> c_ushort(-3)
c_ushort(65533)
>>>
```

Ces types étant des mutables, leur valeur peut aussi être modifiée après coup :

```
>>> i = c_int(42)
>>> print(i)
c_long(42)
>>> print(i.value)
42
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> i.value = -99
>>> print(i.value)
-99
>>>
```

Affecter une nouvelle valeur à une instance de type pointeur — `c_char_p`, `c_wchar_p` et `c_void_p` — change *la zone mémoire* sur laquelle elle pointe, et non *le contenu* de ce bloc mémoire (c'est logique parce que les objets octets sont immuables en Python) :

```
>>> s = "Hello, World"
>>> c_s = c_wchar_p(s)
>>> print(c_s)
c_wchar_p(139966785747344)
>>> print(c_s.value)
Hello World
>>> c_s.value = "Hi, there"
>>> print(c_s)               # the memory location has changed
c_wchar_p(139966783348904)
>>> print(c_s.value)
Hi, there
>>> print(s)                 # first object is unchanged
Hello, World
>>>
```

Cependant, prenez garde à ne pas en passer à des fonctions qui prennent en paramètre des pointeurs sur de la mémoire modifiable. S'il vous faut de la mémoire modifiable, *ctypes* fournit la fonction `create_string_buffer()` qui en crée de plusieurs façons. L'attribut `raw` permet d'accéder à (ou de modifier) un bloc mémoire ; l'attribut `value` permet d'y accéder comme à une chaîne de caractères terminée par NUL :

```
>>> from ctypes import *
>>> p = create_string_buffer(3)            # create a 3 byte buffer, initialized␣
↪to NUL bytes
>>> print(sizeof(p), repr(p.raw))
3 b'\x00\x00\x00'
>>> p = create_string_buffer(b"Hello")      # create a buffer containing a NUL␣
↪terminated string
>>> print(sizeof(p), repr(p.raw))
6 b'Hello\x00'
>>> print(repr(p.value))
b'Hello'
>>> p = create_string_buffer(b"Hello", 10) # create a 10 byte buffer
>>> print(sizeof(p), repr(p.raw))
10 b'Hello\x00\x00\x00\x00\x00'
>>> p.value = b"Hi"
>>> print(sizeof(p), repr(p.raw))
10 b'Hi\x00lo\x00\x00\x00\x00\x00'
>>>
```

The `create_string_buffer()` function replaces the old `c_buffer()` function (which is still available as an alias). To create a mutable memory block containing unicode characters of the C type `wchar_t`, use the `create_unicode_buffer()` function.

Appel de fonctions, suite

printf utilise la vraie sortie standard, et non `sys.stdout` ; les exemples suivants ne fonctionnent donc que dans une invite de commande et non depuis *IDLE* or *PythonWin* :

```
>>> printf = libc.printf
>>> printf(b"Hello, %s\n", b"World!")
Hello, World!
14
>>> printf(b"Hello, %S\n", "World!")
Hello, World!
14
>>> printf(b"%d bottles of beer\n", 42)
42 bottles of beer
19
>>> printf(b"%f bottles of beer\n", 42.5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ctypes.ArgumentError: argument 2: TypeError: Don't know how to convert parameter 2
>>>
```

Comme mentionné plus haut, tous les types Python (les entiers, les chaînes de caractères et les objets octet exceptés) doivent être encapsulés dans leur type *ctypes* correspondant pour pouvoir être convertis dans le type C requis :

```
>>> printf(b"An int %d, a double %f\n", 1234, c_double(3.14))
An int 1234, a double 3.140000
31
>>>
```

Calling variadic functions

On a lot of platforms calling variadic functions through ctypes is exactly the same as calling functions with a fixed number of parameters. On some platforms, and in particular ARM64 for Apple Platforms, the calling convention for variadic functions is different than that for regular functions.

On those platforms it is required to specify the *argtypes* attribute for the regular, non-variadic, function arguments :

```
libc.printf.argtypes = [ctypes.c_char_p]
```

Because specifying the attribute does not inhibit portability it is advised to always specify *argtypes* for all variadic functions.

Appel de fonctions avec des types de données personnalisés

You can also customize *ctypes* argument conversion to allow instances of your own classes be used as function arguments. *ctypes* looks for an `_as_parameter_` attribute and uses this as the function argument. The attribute must be an integer, string, bytes, a *ctypes* instance, or an object with an `_as_parameter_` attribute :

```
>>> class Bottles:
...     def __init__(self, number):
...         self._as_parameter_ = number
...
>>> bottles = Bottles(42)
>>> printf(b"%d bottles of beer\n", bottles)
42 bottles of beer
19
>>>
```

If you don't want to store the instance's data in the `_as_parameter_` instance variable, you could define a *property* which makes the attribute available on request.

Définition du type des arguments nécessaires (prototypes de fonction)

It is possible to specify the required argument types of functions exported from DLLs by setting the *argtypes* attribute.

argtypes must be a sequence of C data types (the `printf()` function is probably not a good example here, because it takes a variable number and different types of parameters depending on the format string, on the other hand this is quite handy to experiment with this feature) :

```
>>> printf.argtypes = [c_char_p, c_char_p, c_int, c_double]
>>> printf(b"String '%s', Int %d, Double %f\n", b"Hi", 10, 2.2)
String 'Hi', Int 10, Double 2.200000
37
>>>
```

Définir un format empêche de passer des arguments de type incompatible (comme le fait le prototype d'une fonction C) et tente de convertir les arguments en des types valides :

```
>>> printf(b"%d %d %d", 1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ctypes.ArgumentError: argument 2: TypeError: 'int' object cannot be interpreted as␣
↪ctypes.c_char_p
>>> printf(b"%s %d %f\n", b"X", 2, 3)
X 2 3.000000
13
>>>
```

If you have defined your own classes which you pass to function calls, you have to implement a *from_param()* class method for them to be able to use them in the *argtypes* sequence. The *from_param()* class method receives the Python object passed to the function call, it should do a typecheck or whatever is needed to make sure this object is acceptable, and then return the object itself, its `_as_parameter_` attribute, or whatever you want to pass as the C function argument in this case. Again, the result should be an integer, string, bytes, a *ctypes* instance, or an object with an `_as_parameter_` attribute.

Types de sortie

By default functions are assumed to return the C `int` type. Other return types can be specified by setting the *restype* attribute of the function object.

The C prototype of `time()` is `time_t time(time_t *)`. Because `time_t` might be of a different type than the default return type `int`, you should specify the `restype` attribute :

```
>>> libc.time.restype = c_time_t
```

The argument types can be specified using *argtypes* :

```
>>> libc.time.argtypes = (POINTER(c_time_t),)
```

To call the function with a `NULL` pointer as first argument, use `None` :

```
>>> print(libc.time(None))
1150640792
```

Here is a more advanced example, it uses the `strchr()` function, which expects a string pointer and a char, and returns a pointer to a string :

```
>>> strchr = libc.strchr
>>> strchr(b"abcdef", ord("d"))
8059983
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> strchr.restype = c_char_p     # c_char_p is a pointer to a string
>>> strchr(b"abcdef", ord("d"))
b'def'
>>> print(strchr(b"abcdef", ord("x")))
None
>>>
```

If you want to avoid the `ord("x")` calls above, you can set the `argtypes` attribute, and the second argument will be converted from a single character Python bytes object into a C char :

```
>>> strchr.restype = c_char_p
>>> strchr.argtypes = [c_char_p, c_char]
>>> strchr(b"abcdef", b"d")
b'def'
>>> strchr(b"abcdef", b"def")
Traceback (most recent call last):
ctypes.ArgumentError: argument 2: TypeError: one character bytes, bytearray or␣
↪integer expected
>>> print(strchr(b"abcdef", b"x"))
None
>>> strchr(b"abcdef", b"d")
b'def'
>>>
```

You can also use a callable Python object (a function or a class for example) as the `restype` attribute, if the foreign function returns an integer. The callable will be called with the *integer* the C function returns, and the result of this call will be used as the result of your function call. This is useful to check for error return values and automatically raise an exception :

```
>>> GetModuleHandle = windll.kernel32.GetModuleHandleA
>>> def ValidHandle(value):
...     if value == 0:
...         raise WinError()
...     return value
...
>>>
>>> GetModuleHandle.restype = ValidHandle
>>> GetModuleHandle(None)
486539264
>>> GetModuleHandle("something silly")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in ValidHandle
OSError: [Errno 126] The specified module could not be found.
>>>
```

`WinError` appelle l'API Windows `FormatMessage()` pour obtenir une représentation de la chaîne de caractères qui correspond au code d'erreur, et *renvoie* une exception. `WinError` prend en paramètre — optionnel — le code d'erreur. Si celui-ci n'est pas passé, elle appelle `GetLastError()` pour le récupérer.

Please note that a much more powerful error checking mechanism is available through the `errcheck` attribute ; see the reference manual for details.

Passage de pointeurs (passage de paramètres par référence)

Il arrive qu'une fonction C du code à interfacer requière un *pointeur* vers un certain type de donnée en paramètre, typiquement pour écrire à l'endroit correspondant ou si la donnée est trop grande pour pouvoir être passée par valeur. Ce mécanisme est appelé *passage de paramètres par référence*.

`ctypes` contient la fonction `byref()` qui permet de passer des paramètres par référence. La fonction `pointer()` a la même utilité, mais fait plus de travail car `pointer()` construit un véritable objet pointeur. Ainsi, si vous n'avez pas besoin de cet objet dans votre code Python, utiliser `byref()` est plus performant :

```
>>> i = c_int()
>>> f = c_float()
>>> s = create_string_buffer(b'\000' * 32)
>>> print(i.value, f.value, repr(s.value))
0 0.0 b''
>>> libc.sscanf(b"1 3.14 Hello", b"%d %f %s",
...             byref(i), byref(f), s)
3
>>> print(i.value, f.value, repr(s.value))
1 3.1400001049 b'Hello'
>>>
```

Structures et unions

Structures and unions must derive from the `Structure` and `Union` base classes which are defined in the `ctypes` module. Each subclass must define a `_fields_` attribute. `_fields_` must be a list of *2-tuples*, containing a *field name* and a *field type*.

Le type de champ doit être un type `ctypes` comme `c_int` ou un type `ctypes` dérivé : structure, union, tableau ou pointeur.

Voici un exemple simple : une structure POINT qui contient deux entiers *x* et *y* et qui montre également comment instancier une structure avec le constructeur :

```
>>> from ctypes import *
>>> class POINT(Structure):
...     _fields_ = [("x", c_int),
...                 ("y", c_int)]
...
>>> point = POINT(10, 20)
>>> print(point.x, point.y)
10 20
>>> point = POINT(y=5)
>>> print(point.x, point.y)
0 5
>>> POINT(1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: too many initializers
>>>
```

Il est bien entendu possible de créer des structures plus complexes. Une structure peut elle-même contenir d'autres structures en prenant une structure comme type de champ.

Voici une structure RECT qui contient deux POINTs *upperleft* et *lowerright* :

```
>>> class RECT(Structure):
...     _fields_ = [("upperleft", POINT),
...                 ("lowerright", POINT)]
...
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> rc = RECT(point)
>>> print(rc.upperleft.x, rc.upperleft.y)
0 5
>>> print(rc.lowerright.x, rc.lowerright.y)
0 0
>>>
```

Une structure encapsulée peut être instanciée par un constructeur de plusieurs façons :

```
>>> r = RECT(POINT(1, 2), POINT(3, 4))
>>> r = RECT((1, 2), (3, 4))
```

Il est possible de récupérer les *descripteurs* des champs depuis la *classe*. Ils sont importants pour déboguer car ils contiennent des informations utiles :

```
>>> print(POINT.x)
<Field type=c_long, ofs=0, size=4>
>>> print(POINT.y)
<Field type=c_long, ofs=4, size=4>
>>>
```

Avertissement

`ctypes` ne prend pas en charge le passage par valeur des unions ou des structures avec des champs de bits. Bien que cela puisse fonctionner sur des architectures 32 bits avec un jeu d'instructions x86, ce n'est pas garanti par la bibliothèque en général. Les unions et les structures avec des champs de bits doivent toujours être passées par pointeur.

Structure/union layout, alignment and byte order

By default, Structure and Union fields are laid out in the same way the C compiler does it. It is possible to override this behavior entirely by specifying a `_layout_` class attribute in the subclass definition ; see the attribute documentation for details.

It is possible to specify the maximum alignment for the fields by setting the `_pack_` class attribute to a positive integer. This matches what `#pragma pack(n)` does in MSVC.

It is also possible to set a minimum alignment for how the subclass itself is packed in the same way `#pragma align(n)` works in MSVC. This can be achieved by specifying a :`_align_` class attribute in the subclass definition.

`ctypes` suit le boutisme natif pour les *Structure* et les *Union*. Pour construire des structures avec un boutisme différent, utilisez les classes de base `BigEndianStructure`, `LittleEndianStructure`, `BigEndianUnion` ou `LittleEndianUnion`. Ces classes ne peuvent pas avoir de champ pointeur.

Champs de bits dans les structures et les unions

It is possible to create structures and unions containing bit fields. Bit fields are only possible for integer fields, the bit width is specified as the third item in the `_fields_` tuples :

```
>>> class Int(Structure):
...     _fields_ = [("first_16", c_int, 16),
...                 ("second_16", c_int, 16)]
...
>>> print(Int.first_16)
<Field type=c_long, ofs=0:0, bits=16>
>>> print(Int.second_16)
```

(suite sur la page suivante)

(suite de la page précédente)

```
<Field type=c_long, ofs=0:16, bits=16>
>>>
```

Tableaux

Les tableaux sont des séquences qui contiennent un nombre fixe d'instances du même type.

La meilleure façon de créer des tableaux consiste à multiplier le type de donnée par un entier positif :

```
TenPointsArrayType = POINT * 10
```

Voici un exemple — un peu artificiel — d'une structure contenant, entre autres, 4 POINTs :

```
>>> from ctypes import *
>>> class POINT(Structure):
...     _fields_ = ("x", c_int), ("y", c_int)
...
>>> class MyStruct(Structure):
...     _fields_ = [("a", c_int),
...                 ("b", c_float),
...                 ("point_array", POINT * 4)]
>>>
>>> print(len(MyStruct().point_array))
4
>>>
```

Comme d'habitude, on crée les instances en appelant la classe :

```
arr = TenPointsArrayType()
for pt in arr:
    print(pt.x, pt.y)
```

Le code précédent affiche une suite de `0 0` car le contenu du tableau est initialisé avec des zéros.

Des valeurs d'initialisation du bon type peuvent être passées :

```
>>> from ctypes import *
>>> TenIntegers = c_int * 10
>>> ii = TenIntegers(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
>>> print(ii)
<c_long_Array_10 object at 0x...>
>>> for i in ii: print(i, end=" ")
...
1 2 3 4 5 6 7 8 9 10
>>>
```

Pointeurs

On crée une instance de pointeur en appelant la fonction *pointer()* sur un type *ctypes* :

```
>>> from ctypes import *
>>> i = c_int(42)
>>> pi = pointer(i)
>>>
```

Les instances de pointeurs ont un attribut *contents* qui renvoie l'objet pointé (l'objet `i` ci-dessus) :

```
>>> pi.contents
c_long(42)
>>>
```

Attention, `ctypes` ne fait pas de ROI (retour de l'objet initial). Il crée un nouvel objet à chaque fois qu'on accède à un attribut :

```
>>> pi.contents is i
False
>>> pi.contents is pi.contents
False
>>>
```

Affecter une autre instance de `c_int` à l'attribut *contents* du pointeur fait pointer le pointeur vers l'adresse mémoire de cette nouvelle instance :

```
>>> i = c_int(99)
>>> pi.contents = i
>>> pi.contents
c_long(99)
>>>
```

Il est possible d'indexer les pointeurs par des entiers :

```
>>> pi[0]
99
>>>
```

Affecter à travers un indice change la valeur pointée :

```
>>> print(i)
c_long(99)
>>> pi[0] = 22
>>> print(i)
c_long(22)
>>>
```

Si vous êtes sûr de vous, vous pouvez utiliser d'autres valeurs que 0, comme en C : il est ainsi possible de modifier une zone mémoire de votre choix. De manière générale cette fonctionnalité ne s'utilise que sur un pointeur renvoyé par une fonction C, pointeur que vous *savez* pointer vers un tableau et non sur un seul élément.

Sous le capot, la fonction `pointer()` fait plus que simplement créer une instance de pointeur ; elle doit d'abord créer un type « pointeur sur… ». Cela s'effectue avec la fonction `POINTER()`, qui prend en paramètre n'importe quel type `ctypes` et renvoie un nouveau type :

```
>>> PI = POINTER(c_int)
>>> PI
<class 'ctypes.LP_c_long'>
>>> PI(42)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: expected c_long instead of int
>>> PI(c_int(42))
<ctypes.LP_c_long object at 0x...>
>>>
```

Appeler le pointeur sur type sans arguments crée un pointeur `NULL`. Les pointeurs `NULL` s'évaluent à `False` :

```
>>> null_ptr = POINTER(c_int)()
>>> print(bool(null_ptr))
False
>>>
```

ctypes vérifie que le pointeur n'est pas `NULL` quand il en déréférence un (mais déréférencer des pointeurs non `NULL` invalides fait planter Python) :

```
>>> null_ptr[0]
Traceback (most recent call last):
    ....
ValueError: NULL pointer access
>>>

>>> null_ptr[0] = 1234
Traceback (most recent call last):
    ....
ValueError: NULL pointer access
>>>
```

Thread safety without the GIL

In Python 3.13, the *GIL* may be disabled on *experimental free threaded* builds. In ctypes, reads and writes to a single object concurrently is safe, but not across multiple objects :

```
>>> number = c_int(42)
>>> pointer_a = pointer(number)
>>> pointer_b = pointer(number)
```

In the above, it's only safe for one object to read and write to the address at once if the GIL is disabled. So, `pointer_a` can be shared and written to across multiple threads, but only if `pointer_b` is not also attempting to do the same. If this is an issue, consider using a *threading.Lock* to synchronize access to memory :

```
>>> import threading
>>> lock = threading.Lock()
>>> # Thread 1
>>> with lock:
...     pointer_a.contents = 24
>>> # Thread 2
>>> with lock:
...     pointer_b.contents = 42
```

Conversions de type

Usually, ctypes does strict type checking. This means, if you have `POINTER(c_int)` in the *argtypes* list of a function or as the type of a member field in a structure definition, only instances of exactly the same type are accepted. There are some exceptions to this rule, where ctypes accepts other objects. For example, you can pass compatible array instances instead of pointer types. So, for `POINTER(c_int)`, ctypes accepts an array of c_int :

```
>>> class Bar(Structure):
...     _fields_ = [("count", c_int), ("values", POINTER(c_int))]
...
>>> bar = Bar()
>>> bar.values = (c_int * 3)(1, 2, 3)
>>> bar.count = 3
>>> for i in range(bar.count):
...     print(bar.values[i])
```

(suite sur la page suivante)

(suite de la page précédente)

```
...
1
2
3
>>>
```

In addition, if a function argument is explicitly declared to be a pointer type (such as `POINTER(c_int)`) in *argtypes*, an object of the pointed type (`c_int` in this case) can be passed to the function. ctypes will apply the required *byref()* conversion in this case automatically.

Pour mettre un champ de type *POINTER* à `NULL`, il faut lui affecter `None` :

```
>>> bar.values = None
>>>
```

Parfois il faut gérer des incompatibilités entre les types. En C, il est possible de convertir un type en un autre. *ctypes* fournit la fonction *cast()* qui permet la même chose. La structure `Bar` ci-dessus accepte des pointeurs `POINTER(c_int)` ou des tableaux de *c_int* comme valeur pour le champ `values`, mais pas des instances d'autres types :

```
>>> bar.values = (c_byte * 4)()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: incompatible types, c_byte_Array_4 instance instead of LP_c_long␣
↪instance
>>>
```

C'est là que la fonction *cast()* intervient.

La fonction *cast()* permet de convertir une instance de *ctypes* en un pointeur vers un type de données *ctypes* différent. *cast()* prend deux paramètres : un objet *ctypes* qui est, ou qui peut être converti en, un certain pointeur et un type pointeur de *ctypes*. Elle renvoie une instance du second argument, qui pointe sur le même bloc mémoire que le premier argument :

```
>>> a = (c_byte * 4)()
>>> cast(a, POINTER(c_int))
<ctypes.LP_c_long object at ...>
>>>
```

Ainsi, la fonction *cast()* permet de remplir le champ `values` de la structure `Bar` :

```
>>> bar = Bar()
>>> bar.values = cast((c_byte * 4)(), POINTER(c_int))
>>> print(bar.values[0])
0
>>>
```

Types incomplets

Un *type incomplet* est une structure, une union ou un tableau dont les membres ne sont pas encore définis. C'est l'équivalent d'une déclaration avancée en C, où la définition est fournie plus tard :

```
struct cell; /* forward declaration */

struct cell {
    char *name;
    struct cell *next;
};
```

Une traduction naïve, mais invalide, en code *ctypes* ressemblerait à ça :

```
>>> class cell(Structure):
...     _fields_ = [("name", c_char_p),
...                 ("next", POINTER(cell))]
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in cell
NameError: name 'cell' is not defined
>>>
```

because the new `class cell` is not available in the class statement itself. In *ctypes*, we can define the `cell` class and set the *_fields_* attribute later, after the class statement :

```
>>> from ctypes import *
>>> class cell(Structure):
...     pass
...
>>> cell._fields_ = [("name", c_char_p),
...                  ("next", POINTER(cell))]
>>>
```

Essayons. Nous créons deux instances de `cell`, les faisons pointer l'une sur l'autre et enfin nous suivons quelques maillons de la chaîne de pointeurs :

```
>>> c1 = cell()
>>> c1.name = b"foo"
>>> c2 = cell()
>>> c2.name = b"bar"
>>> c1.next = pointer(c2)
>>> c2.next = pointer(c1)
>>> p = c1
>>> for i in range(8):
...     print(p.name, end=" ")
...     p = p.next[0]
...
foo bar foo bar foo bar foo bar
>>>
```

Fonctions de rappel

ctypes permet de créer des pointeurs de fonctions appelables par des appelables Python. On les appelle parfois *fonctions de rappel*.

Tout d'abord, il faut créer une classe pour la fonction de rappel. La classe connaît la convention d'appel, le type de retour ainsi que le nombre et le type de paramètres que la fonction accepte.

La fabrique *CFUNCTYPE()* crée un type pour les fonctions de rappel qui suivent la convention d'appel `cdecl`. En Windows, c'est la fabrique *WINFUNCTYPE()* qui crée un type pour les fonctions de rappel qui suivent la convention d'appel `stdcall`.

Le premier paramètre de ces deux fonctions est le type de retour, et les suivants sont les types des arguments qu'attend la fonction de rappel.

I will present an example here which uses the standard C library's `qsort()` function, that is used to sort items with the help of a callback function. `qsort()` will be used to sort an array of integers :

```
>>> IntArray5 = c_int * 5
>>> ia = IntArray5(5, 1, 7, 33, 99)
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> qsort = libc.qsort
>>> qsort.restype = None
>>>
```

`qsort()` must be called with a pointer to the data to sort, the number of items in the data array, the size of one item, and a pointer to the comparison function, the callback. The callback will then be called with two pointers to items, and it must return a negative integer if the first item is smaller than the second, a zero if they are equal, and a positive integer otherwise.

Ainsi notre fonction de rappel reçoit des pointeurs vers des entiers et doit renvoyer un entier. Créons d'abord le `type` pour la fonction de rappel :

```
>>> CMPFUNC = CFUNCTYPE(c_int, POINTER(c_int), POINTER(c_int))
>>>
```

Pour commencer, voici une fonction de rappel simple qui affiche les valeurs qu'on lui passe :

```
>>> def py_cmp_func(a, b):
...     print("py_cmp_func", a[0], b[0])
...     return 0
...
>>> cmp_func = CMPFUNC(py_cmp_func)
>>>
```

Résultat :

```
>>> qsort(ia, len(ia), sizeof(c_int), cmp_func)
py_cmp_func 5 1
py_cmp_func 33 99
py_cmp_func 7 33
py_cmp_func 5 7
py_cmp_func 1 7
>>>
```

À présent, comparons pour de vrai les deux entiers et renvoyons un résultat utile :

```
>>> def py_cmp_func(a, b):
...     print("py_cmp_func", a[0], b[0])
...     return a[0] - b[0]
...
>>>
>>> qsort(ia, len(ia), sizeof(c_int), CMPFUNC(py_cmp_func))
py_cmp_func 5 1
py_cmp_func 33 99
py_cmp_func 7 33
py_cmp_func 1 7
py_cmp_func 5 7
>>>
```

Et comme il est facile de le voir, notre tableau est désormais classé :

```
>>> for i in ia: print(i, end=" ")
...
1 5 7 33 99
>>>
```

Ces fonctions peuvent aussi être utilisées comme des décorateurs ; il est donc possible d'écrire :

```
>>> @CFUNCTYPE(c_int, POINTER(c_int), POINTER(c_int))
... def py_cmp_func(a, b):
...     print("py_cmp_func", a[0], b[0])
...     return a[0] - b[0]
...
>>> qsort(ia, len(ia), sizeof(c_int), py_cmp_func)
py_cmp_func 5 1
py_cmp_func 33 99
py_cmp_func 7 33
py_cmp_func 1 7
py_cmp_func 5 7
>>>
```

Note

Prenez garde à bien conserver une référence à un objet `CFUNCTYPE()` tant que celui-ci est utilisé par le code C. `ctypes` ne le fait pas tout seul et, si vous ne le faites pas, le ramasse-miette pourrait les libérer, ce qui fera planter votre programme quand un appel sera fait.

Notez aussi que si la fonction de rappel est appelée dans un fil d'exécution créé hors de Python (p. ex. par du code externe qui appelle la fonction de rappel), *ctypes* crée un nouveau fil Python « creux » à chaque fois. Ce comportement est acceptable pour la plupart des cas d'utilisation, mais cela implique que les valeurs stockées avec `threading.local` ne seront *pas* persistantes d'un appel à l'autre, même si les appels proviennent du même fil d'exécution C.

Accès aux variables exportées depuis une DLL

Some shared libraries not only export functions, they also export variables. An example in the Python library itself is the `Py_Version`, Python runtime version number encoded in a single constant integer.

`ctypes` can access values like this with the `in_dll()` class methods of the type. *pythonapi* is a predefined symbol giving access to the Python C api :

```
>>> version = ctypes.c_int.in_dll(ctypes.pythonapi, "Py_Version")
>>> print(hex(version.value))
0x30c00a0
```

Le pointeur `PyImport_FrozenModules` exposé par Python est un autre exemple complet de l'utilisation de pointeurs.

Citons la documentation :

> This pointer is initialized to point to an array of `_frozen` records, terminated by one whose members are all `NULL` or zero. When a frozen module is imported, it is searched in this table. Third-party code could play tricks with this to provide a dynamically created collection of frozen modules.

Donc manipuler ce pointeur peut même se révéler utile. Pour limiter la taille de l'exemple, nous nous bornons à montrer comment lire ce tableau avec `ctypes` :

```
>>> from ctypes import *
>>>
>>> class struct_frozen(Structure):
...     _fields_ = [("name", c_char_p),
...                 ("code", POINTER(c_ubyte)),
...                 ("size", c_int),
...                 ("get_code", POINTER(c_ubyte)),  # Function pointer
...                ]
...
>>>
```

We have defined the `_frozen` data type, so we can get the pointer to the table :

```
>>> FrozenTable = POINTER(struct_frozen)
>>> table = FrozenTable.in_dll(pythonapi, "_PyImport_FrozenBootstrap")
>>>
```

Puisque `table` est un `pointer` vers un tableau d'entrées de `struct_frozen`, il est possible d'itérer dessus, mais il faut être certain que la boucle se termine, car les pointeurs n'ont pas de taille. Tôt ou tard, il planterait probablement avec une erreur de segmentation ou autre, donc mieux vaut sortir de la boucle quand on lit l'entrée `NULL` :

```
>>> for item in table:
...     if item.name is None:
...         break
...     print(item.name.decode("ascii"), item.size)
...
_frozen_importlib 31764
_frozen_importlib_external 41499
zipimport 12345
>>>
```

Le fait que le standard Python possède un module et un paquet figés (indiqués par la valeur négative du membre *size*) est peu connu, cela ne sert qu'aux tests. Essayez avec `import __hello__` par exemple.

Pièges

Il y a quelques cas tordus dans *ctypes* où on peut s'attendre à un résultat différent de la réalité.

Examinons l'exemple suivant :

```
>>> from ctypes import *
>>> class POINT(Structure):
...     _fields_ = ("x", c_int), ("y", c_int)
...
>>> class RECT(Structure):
...     _fields_ = ("a", POINT), ("b", POINT)
...
>>> p1 = POINT(1, 2)
>>> p2 = POINT(3, 4)
>>> rc = RECT(p1, p2)
>>> print(rc.a.x, rc.a.y, rc.b.x, rc.b.y)
1 2 3 4
>>> # now swap the two points
>>> rc.a, rc.b = rc.b, rc.a
>>> print(rc.a.x, rc.a.y, rc.b.x, rc.b.y)
3 4 3 4
>>>
```

Diantre. On s'attendait certainement à ce que le dernier résultat affiche `3 4 1 2`. Que s'est-il passé ? Les étapes de la ligne `rc.a, rc.b = rc.b, rc.a` ci-dessus sont les suivantes :

```
>>> temp0, temp1 = rc.b, rc.a
>>> rc.a = temp0
>>> rc.b = temp1
>>>
```

Les objets `temp0` et `temp1` utilisent encore le tampon interne de l'objet `rc` ci-dessus. Donc exécuter `rc.a = temp0` copie le contenu du tampon de `temp0` dans celui de `rc`. Ce qui, par ricochet, modifie le contenu de `temp1`. Et donc, la dernière affectation, `rc.b = temp1`, n'a pas l'effet escompté.

Gardez en tête qu'accéder au sous-objet depuis une *Structure*, une *Union* ou un *Array* ne copie *pas* le sous-objet, mais crée un objet interface qui accède au tampon sous-jacent de l'objet initial.

Un autre exemple de comportement *a priori* inattendu est le suivant :

```
>>> s = c_char_p()
>>> s.value = b"abc def ghi"
>>> s.value
b'abc def ghi'
>>> s.value is s.value
False
>>>
```

Note

La valeur d'une instance de `c_char_p` ne peut être initialisée qu'avec un octet ou un entier.

Pourquoi cela affiche-t'il `False` ? Les instances *ctypes* sont des objets qui contiennent un bloc mémoire et des *descriptor* qui donnent accès au contenu du ce bloc. Stocker un objet Python dans le bloc mémoire ne stocke pas l'objet même ; seuls ses `contents` le sont. Accéder au `contents` crée un nouvel objet Python à chaque fois !

Types de données à taille flottante

`ctypes` assure la prise en charge des tableaux et des structures à taille flottante.

La fonction `resize()` permet de redimensionner la taille du tampon mémoire d'un objet *ctypes* existant. Cette fonction prend l'objet comme premier argument et la taille en octets désirée comme second. La taille du tampon mémoire ne peut pas être inférieure à celle occupée par un objet unitaire du type considéré. Une `ValueError` est levée si c'est le cas :

```
>>> short_array = (c_short * 4)()
>>> print(sizeof(short_array))
8
>>> resize(short_array, 4)
Traceback (most recent call last):
    ...
ValueError: minimum size is 8
>>> resize(short_array, 32)
>>> sizeof(short_array)
32
>>> sizeof(type(short_array))
8
>>>
```

Cela dit, comment accéder aux éléments supplémentaires contenus dans le tableau ? Vu que le type ne connaît que 4 éléments, on obtient une erreur si l'on accède aux suivants :

```
>>> short_array[:]
[0, 0, 0, 0]
>>> short_array[7]
Traceback (most recent call last):
    ...
IndexError: invalid index
>>>
```

Une autre approche pour utiliser des types de donnée à taille flottante avec `ctypes` consiste à tirer profit de la nature intrinsèquement dynamique de Python et de (re)définir le type de donnée une fois que la taille demandée est connue, au cas-par-cas.

16.9.2 Référence du module

Recherche de bibliothèques partagées

Les langages compilés ont besoin d'accéder aux bibliothèques partagées au moment de la compilation, de l'édition de liens et pendant l'exécution du programme.

The purpose of the *find_library()* function is to locate a library in a way similar to what the compiler or runtime loader does (on platforms with several versions of a shared library the most recent should be loaded), while the ctypes library loaders act like when a program is run, and call the runtime loader directly.

The `ctypes.util` module provides a function which can help to determine the library to load.

`ctypes.util.`**`find_library`**(*name*)

Tente de trouver une bibliothèque et en renvoie le chemin. *name* est le nom de la bibliothèque sans préfixe — comme *lib* — ni suffixe — comme `.so`, `.dylib` ou un numéro de version (c.-à-d. la même forme que l'option POSIX de l'éditeur de lien `-l`). Si la fonction ne parvient pas à trouver de bibliothèque, elle renvoie `None`.

Le mode opératoire exact dépend du système.

On Linux, *find_library()* tries to run external programs (`/sbin/ldconfig`, `gcc`, `objdump` and `ld`) to find the library file. It returns the filename of the library file.

Modifié dans la version 3.6 : Sous Linux, si les autres moyens échouent, la fonction utilise la variable d'environnement `LD_LIBRARY_PATH` pour trouver la bibliothèque.

Voici quelques exemples :

```
>>> from ctypes.util import find_library
>>> find_library("m")
'libm.so.6'
>>> find_library("c")
'libc.so.6'
>>> find_library("bz2")
'libbz2.so.1.0'
>>>
```

On macOS and Android, *find_library()* uses the system's standard naming schemes and paths to locate the library, and returns a full pathname if successful :

```
>>> from ctypes.util import find_library
>>> find_library("c")
'/usr/lib/libc.dylib'
>>> find_library("m")
'/usr/lib/libm.dylib'
>>> find_library("bz2")
'/usr/lib/libbz2.dylib'
>>> find_library("AGL")
'/System/Library/Frameworks/AGL.framework/AGL'
>>>
```

On Windows, *find_library()* searches along the system search path, and returns the full pathname, but since there is no predefined naming scheme a call like `find_library("c")` will fail and return `None`.

If wrapping a shared library with *ctypes*, it *may* be better to determine the shared library name at development time, and hardcode that into the wrapper module instead of using *find_library()* to locate the library at runtime.

Listing loaded shared libraries

When writing code that relies on code loaded from shared libraries, it can be useful to know which shared libraries have already been loaded into the current process.

The `ctypes.util` module provides the *dllist()* function, which calls the different APIs provided by the various platforms to help determine which shared libraries have already been loaded into the current process.

The exact output of this function will be system dependent. On most platforms, the first entry of this list represents the current process itself, which may be an empty string. For example, on glibc-based Linux, the return may look like :

```
>>> from ctypes.util import dllist
>>> dllist()
['', 'linux-vdso.so.1', '/lib/x86_64-linux-gnu/libm.so.6', '/lib/x86_64-linux-gnu/
↪libc.so.6', ... ]
```

Chargement des bibliothèques partagées

Il y a plusieurs moyens de charger une bibliothèque partagée dans un processus Python. L'un d'entre eux consiste à instancier une des classes suivantes :

class ctypes.**CDLL**(*name*, *mode=DEFAULT_MODE*, *handle=None*, *use_errno=False*, *use_last_error=False*, *winmode=None*)

Instances of this class represent loaded shared libraries. Functions in these libraries use the standard C calling convention, and are assumed to return `int`.

En Windows, créer une instance de `CDLL` peut échouer, même si une DLL du bon nom existe. Quand une des dépendances de la DLL à charger ne peut pas être trouvée, une `OSError` est levée avec le message *"[WinError 126] The specified module could not be found"*. Ce message d'erreur ne contient pas le nom de la DLL manquante car l'API Windows ne fournit pas cette information. Cela rend l'erreur délicate à analyser ; pour la résoudre, il faut lister toutes les dépendances de la DLL et trouver celle qui manque en utilisant des outils de débogage et de traçage Windows.

Modifié dans la version 3.12 : The *name* parameter can now be a *path-like object*.

Voir aussi

DUMPBIN — un utilitaire Microsoft pour lister les dépendances d'une DLL.

class ctypes.**OleDLL**(*name*, *mode=DEFAULT_MODE*, *handle=None*, *use_errno=False*, *use_last_error=False*, *winmode=None*)

Instances of this class represent loaded shared libraries, functions in these libraries use the `stdcall` calling convention, and are assumed to return the windows specific `HRESULT` code. `HRESULT` values contain information specifying whether the function call failed or succeeded, together with additional error code. If the return value signals a failure, an `OSError` is automatically raised.

Availability : Windows

Modifié dans la version 3.3 : `WindowsError` used to be raised, which is now an alias of `OSError`.

Modifié dans la version 3.12 : The *name* parameter can now be a *path-like object*.

class ctypes.**WinDLL**(*name*, *mode=DEFAULT_MODE*, *handle=None*, *use_errno=False*, *use_last_error=False*, *winmode=None*)

Instances of this class represent loaded shared libraries, functions in these libraries use the `stdcall` calling convention, and are assumed to return `int` by default.

Availability : Windows

Modifié dans la version 3.12 : The *name* parameter can now be a *path-like object*.

Le *verrou global de l'interpréteur* Python est relâché avant chaque appel d'une fonction exposée par ces bibliothèques et ré-activé après.

class ctypes.**PyDLL**(*name*, *mode=DEFAULT_MODE*, *handle=None*)

Cette classe est identique à `CDLL`, à ceci près que le GIL n'est *pas* relâché pendant l'appel de la fonction, et, qu'au terme de l'appel, le drapeau d'erreur Python est vérifié. Si celui-ci est activé, une exception Python est levée.

Donc, cette classe ne sert qu'à appeler les fonctions de l'API C de Python.

Modifié dans la version 3.12 : The *name* parameter can now be a *path-like object*.

All these classes can be instantiated by calling them with at least one argument, the pathname of the shared library. If you have an existing handle to an already loaded shared library, it can be passed as the `handle` named parameter, otherwise the underlying platform's `dlopen()` or `LoadLibrary()` function is used to load the library into the process, and to get a handle to it.

Le mode de chargement de la bibliothèque est défini par le paramètre *mode*. Pour plus de détails, référez-vous à l'entrée *dlopen(3)* du manuel. En Windows, *mode* est ignoré. Sur les systèmes POSIX, RTLD_NOW y est toujours ajouté. Ceci n'est pas configurable.

The *use_errno* parameter, when set to true, enables a ctypes mechanism that allows accessing the system *errno* error number in a safe way. *ctypes* maintains a thread-local copy of the system's *errno* variable ; if you call foreign functions created with `use_errno=True` then the *errno* value before the function call is swapped with the ctypes private copy, the same happens immediately after the function call.

La fonction *ctypes.get_errno()* renvoie la valeur de la copie privée de *ctypes*. La fonction *ctypes.set_errno()* affecte une nouvelle valeur à la copie privée et renvoie l'ancienne valeur.

The *use_last_error* parameter, when set to true, enables the same mechanism for the Windows error code which is managed by the *GetLastError()* and `SetLastError()` Windows API functions ; *ctypes.get_last_error()* and *ctypes.set_last_error()* are used to request and change the ctypes private copy of the windows error code.

The *winmode* parameter is used on Windows to specify how the library is loaded (since *mode* is ignored). It takes any value that is valid for the Win32 API `LoadLibraryEx` flags parameter. When omitted, the default is to use the flags that result in the most secure DLL load, which avoids issues such as DLL hijacking. Passing the full path to the DLL is the safest way to ensure the correct library and dependencies are loaded.

Modifié dans la version 3.8 : Ajout du paramètre *winmode*.

`ctypes.`**`RTLD_GLOBAL`**

Valeur possible pour le paramètre *mode*. Vaut zéro sur les plates-formes où ce drapeau n'est pas disponible.

`ctypes.`**`RTLD_LOCAL`**

Valeur possible pour le paramètre *mode*. Vaut *RTLD_GLOBAL* sur les plates-formes où ce drapeau n'est pas disponible.

`ctypes.`**`DEFAULT_MODE`**

Mode de chargement par défaut des bibliothèques partagées. Vaut *RTLD_GLOBAL* sur OSX 10.3 et *RTLD_LOCAL* sur les autres systèmes d'exploitation.

Les instances de ces classes n'ont pas de méthodes publiques ; on accède aux fonctions de la bibliothèque partagée par attribut ou par indiçage. Notez que les résultats des accès par attribut sont mis en cache, et donc des accès consécutifs renvoient à chaque fois le même objet. Accéder à une fonction par indice renvoie cependant chaque fois un nouvel objet :

```
>>> from ctypes import CDLL
>>> libc = CDLL("libc.so.6")  # On Linux
>>> libc.time == libc.time
True
>>> libc['time'] == libc['time']
False
```

Les attributs publics suivants sont disponibles, leur nom commence par un tiret bas pour éviter les conflits avec les noms des fonctions exportées :

`PyDLL.`**`_handle`**

Le lien système d'accès à la bibliothèque.

`PyDLL.`**`_name`**

Nom de la bibliothèque donné au constructeur.

Shared libraries can also be loaded by using one of the prefabricated objects, which are instances of the *LibraryLoader* class, either by calling the *LoadLibrary()* method, or by retrieving the library as attribute of the loader instance.

class ctypes.**LibraryLoader**(*dlltype*)

Classe pour charger une bibliothèque partagée. *dlltype* doit être de type *CDLL*, *PyDLL*, *WinDLL* ou *OleDLL*.

__getattr__() has special behavior : It allows loading a shared library by accessing it as attribute of a library loader instance. The result is cached, so repeated attribute accesses return the same library each time.

LoadLibrary(*name*)

Charge une bibliothèque partagée dans le processus et la renvoie. Cette méthode renvoie toujours une nouvelle instance de la bibliothèque.

Plusieurs chargeurs sont fournis :

ctypes.**cdll**

Pour créer des instances de *CDLL*.

ctypes.**windll**

Creates *WinDLL* instances.

Availability : Windows

ctypes.**oledll**

Creates *OleDLL* instances.

Availability : Windows

ctypes.**pydll**

Pour créer des instances de *PyDLL*.

Il existe un moyen rapide d'accéder directement à l'API C Python :

ctypes.**pythonapi**

An instance of *PyDLL* that exposes Python C API functions as attributes. Note that all these functions are assumed to return C `int`, which is of course not always the truth, so you have to assign the correct `restype` attribute to use these functions.

Charger une bibliothèque à l'aide d'un de ces objets avec en argument `name`, le nom de la bibliothèque (une chaîne de caractères), lève un *évènement d'audit* `ctypes.dlopen`.

Accéder à une fonction d'une bibliothèque lève un évènement d'audit `ctypes.dlsym` avec `library` (l'objet bibliothèque) et `name` (le nom du symbole — une chaîne de caractères ou un entier) comme arguments.

Si seul le lien sur la bibliothèque, et non l'objet, est disponible, accéder à une fonction lève l'évènement d'audit `ctypes.dlsym/handle` avec `handle` (le lien vers la bibliothèque) et `name` comme arguments.

Fonctions externes

As explained in the previous section, foreign functions can be accessed as attributes of loaded shared libraries. The function objects created in this way by default accept any number of arguments, accept any ctypes data instances as arguments, and return the default result type specified by the library loader.

They are instances of a private local class `_FuncPtr` (not exposed in `ctypes`) which inherits from the private *_CFuncPtr* class :

```
>>> import ctypes
>>> lib = ctypes.CDLL(None)
>>> issubclass(lib._FuncPtr, ctypes._CFuncPtr)
True
>>> lib._FuncPtr is ctypes._CFuncPtr
False
```

class ctypes.**_CFuncPtr**

Classe de base pour les fonctions externes C.

Une instance de fonction externe est également un type de donnée compatible avec le C ; elle représente un pointeur vers une fonction.

Son comportement peut-être personnalisé en réaffectant les attributs spécifiques de l'objet représentant la fonction externe.

restype

Assign a ctypes type to specify the result type of the foreign function. Use `None` for `void`, a function not returning anything.

It is possible to assign a callable Python object that is not a ctypes type, in this case the function is assumed to return a C `int`, and the callable will be called with this integer, allowing further processing or error checking. Using this is deprecated, for more flexible post processing or error checking use a ctypes data type as `restype` and assign a callable to the *errcheck* attribute.

argtypes

Fait correspondre le type des arguments que la fonction accepte avec un *n*-uplet de types *ctypes*. Les fonctions qui utilisent la convention d'appel `stdcall` ne peuvent être appelées qu'avec le même nombre d'arguments que la taille du *n*-uplet mais les fonctions qui utilisent la convention d'appel C acceptent aussi des arguments additionnels non-définis.

When a foreign function is called, each actual argument is passed to the *from_param()* class method of the items in the *argtypes* tuple, this method allows adapting the actual argument to an object that the foreign function accepts. For example, a *c_char_p* item in the *argtypes* tuple will convert a string passed as argument into a bytes object using ctypes conversion rules.

New : It is now possible to put items in argtypes which are not ctypes types, but each item must have a *from_param()* method which returns a value usable as argument (integer, string, ctypes instance). This allows defining adapters that can adapt custom objects as function parameters.

errcheck

Définit une fonction Python ou tout autre appelable qui sera appelé avec trois arguments ou plus :

callable(*result*, *func*, *arguments*)

result is what the foreign function returns, as specified by the `restype` attribute.

func est l'objet représentant la fonction externe elle-même. Cet accesseur permet de réutiliser le même appelable pour vérifier le résultat de plusieurs fonctions ou de faire des actions supplémentaires après leur exécution.

arguments est le *n*-uplet qui contient les paramètres initiaux passés à la fonction, ceci permet de spécialiser le comportement des arguments utilisés.

L'objet renvoyé par cette fonction est celui renvoyé par l'appel de la fonction externe, mais il peut aussi vérifier la valeur du résultat et lever une exception si l'appel a échoué.

On Windows, when a foreign function call raises a system exception (for example, due to an access violation), it will be captured and replaced with a suitable Python exception. Further, an auditing event `ctypes.set_exception` with argument `code` will be raised, allowing an audit hook to replace the exception with its own.

Certaines manières d'appeler des fonction externes peuvent lever des évènements d'audit `ctypes.call_function` avec `function pointer` et `arguments` comme arguments.

Prototypes de fonction

Il est aussi possible de créer des fonctions externes en instanciant des prototypes de fonction. Les prototypes de fonction ressemblent beaucoup aux prototypes de fonctions en C ; ils décrivent une fonction (type de retour, type des arguments, convention d'appel) sans préciser son implémentation. Les fabriques de fonctions prennent en entrée le type de retour et le type des arguments de la fonction, et peuvent être utilisées comme des décorateurs-fabrique et ainsi s'appliquer à des fonctions avec la syntaxe `@décorateur`. Ceci est illustré dans la section *Fonctions de rappel*.

`ctypes.CFUNCTYPE`(*restype*, **argtypes*, *use_errno=False*, *use_last_error=False*)

Renvoie un prototype de fonction qui crée des fonctions qui suivent la convention d'appel standard C. Les fonctions libèreront le GIL lors de leur exécution. Si *use_errno* est vrai, la copie privée *ctypes* de la variable système `errno` est échangée avec la vraie valeur de `errno` avant et après l'appel ; *use_last_error* a le même effet sous Windows.

`ctypes.WINFUNCTYPE`(*restype*, **argtypes*, *use_errno=False*, *use_last_error=False*)

The returned function prototype creates functions that use the `stdcall` calling convention. The function will release the GIL during the call. *use_errno* and *use_last_error* have the same meaning as above.

Availability : Windows

`ctypes.PYFUNCTYPE`(*restype*, **argtypes*)

Renvoie un prototype de fonction qui crée des fonctions qui suivent la convention d'appel Python. Les fonctions ne libèreront *pas* le GIL lors de leur exécution.

Il y a plusieurs façons d'instancier les prototypes de fonction créés par ces fabriques, selon le type et le nombre de paramètres de l'appel :

prototype(*address*)

Renvoie une fonction externe sur l'adresse donnée sous la forme d'un entier.

prototype(*callable*)

Crée une fonction appelable depuis du code C (une fonction de rappel) d'un appelable Python donné en paramètre.

prototype(*func_spec*[, *paramflags*])

Renvoie une fonction externe exposée par une bibliothèque partagée. *func_spec* est un couple `(nom_ou_indice, bibliothèque)`. Le premier élément est le nom de la fonction à passer comme une chaîne ou bien son indice (dans la table des symboles) à passer comme un entier. Le second élément est l'instance de la bibliothèque partagée.

prototype(*vtbl_index*, *name*[, *paramflags*[, *iid*]])

Renvoie une fonction qui appelle une méthode COM. *vtbl_index* est l'indice de la fonction dans la table virtuelle, un petit entier positif. *name* est le nom de la méthode COM. *iid* est un pointeur optionnel vers l'identificateur de plateforme, qui est utilisé dans la remontée d'erreurs étendue.

If *iid* is not specified, an `OSError` is raised if the COM method call fails. If *iid* is specified, a `COMError` is raised instead.

COM methods use a special calling convention : They require a pointer to the COM interface as first argument, in addition to those parameters that are specified in the `argtypes` tuple.

Availability : Windows

Le paramètre optionnel *paramflags* crée une fabrique de fonction externes avec des fonctionnalités supplémentaires par rapport à celles décrites ci-dessus.

paramflags must be a tuple of the same length as `argtypes`.

Chaque élément de ce *n*-uplet contient des informations supplémentaires sur le paramètre correspondant. Ce doit être aussi un *n*-uplet, avec un, deux ou trois éléments.

Le premier élément est un entier qui contient une combinaison de drapeaux qui précisent le sens des paramètres (entrée ou sortie) :

1
Paramètre d'entrée.

2
Paramètre de sortie. La fonction externe va modifier cette valeur.

4
Paramètre d'entrée, valant 0 par défaut.

Le deuxième élément (optionnel) est une chaîne de caractères représentant le nom du paramètre. Si cet élément est donné, la fonction externe pourra être appelée avec des paramètres nommés.

Le troisième élément (optionnel) est la valeur par défaut du paramètre.

The following example demonstrates how to wrap the Windows `MessageBoxW` function so that it supports default parameters and named arguments. The C declaration from the windows header file is this :

```
WINUSERAPI int WINAPI
MessageBoxW(
    HWND hWnd,
    LPCWSTR lpText,
    LPCWSTR lpCaption,
    UINT uType);
```

L'encapsulation *ctypes* correspondante est alors :

```
>>> from ctypes import c_int, WINFUNCTYPE, windll
>>> from ctypes.wintypes import HWND, LPCWSTR, UINT
>>> prototype = WINFUNCTYPE(c_int, HWND, LPCWSTR, LPCWSTR, UINT)
>>> paramflags = (1, "hwnd", 0), (1, "text", "Hi"), (1, "caption", "Hello from␣
↪ctypes"), (1, "flags", 0)
>>> MessageBox = prototype(("MessageBoxW", windll.user32), paramflags)
```

La fonction `MessageBox` peut désormais être appelée des manières suivantes :

```
>>> MessageBox()
>>> MessageBox(text="Spam, spam, spam")
>>> MessageBox(flags=2, text="foo bar")
```

L'exemple qui suit traite des paramètres en sortie. La fonction win32 `GetWindowRect` donne les dimensions d'une fenêtre en les copiant dans une structure `RECT` que l'appelant doit fournir. Sa déclaration en C est :

```
WINUSERAPI BOOL WINAPI
GetWindowRect(
     HWND hWnd,
     LPRECT lpRect);
```

L'encapsulation *ctypes* correspondante est alors :

```
>>> from ctypes import POINTER, WINFUNCTYPE, windll, WinError
>>> from ctypes.wintypes import BOOL, HWND, RECT
>>> prototype = WINFUNCTYPE(BOOL, HWND, POINTER(RECT))
>>> paramflags = (1, "hwnd"), (2, "lprect")
>>> GetWindowRect = prototype(("GetWindowRect", windll.user32), paramflags)
>>>
```

Les fonctions avec des paramètres en sortie renvoient automatiquement la valeur du paramètre de sortie s'il n'y en a qu'un seul, ou un *n*-uplet avec les valeurs de sortie de chaque paramètre s'il y en a plusieurs. Ici, la fonction *GetWindowRect* renvoie donc une instance de *RECT* quand elle est appelée.

Output parameters can be combined with the *errcheck* protocol to do further output processing and error checking. The win32 `GetWindowRect` api function returns a `BOOL` to signal success or failure, so this function could do the error checking, and raises an exception when the api call failed :

```
>>> def errcheck(result, func, args):
...     if not result:
...         raise WinError()
...     return args
...
>>> GetWindowRect.errcheck = errcheck
>>>
```

If the *errcheck* function returns the argument tuple it receives unchanged, *ctypes* continues the normal processing it does on the output parameters. If you want to return a tuple of window coordinates instead of a `RECT` instance, you can retrieve the fields in the function and return them instead, the normal processing will no longer take place :

```
>>> def errcheck(result, func, args):
...     if not result:
...         raise WinError()
...     rc = args[1]
...     return rc.left, rc.top, rc.bottom, rc.right
...
>>> GetWindowRect.errcheck = errcheck
>>>
```

Fonctions utilitaires

`ctypes.`**`addressof`**(*obj*)

Returns the address of the memory buffer as integer. *obj* must be an instance of a ctypes type.

Raises an *auditing event* `ctypes.addressof` with argument `obj`.

`ctypes.`**`alignment`**(*obj_or_type*)

Returns the alignment requirements of a ctypes type. *obj_or_type* must be a ctypes type or instance.

`ctypes.`**`byref`**(*obj*[, *offset*])

Returns a light-weight pointer to *obj*, which must be an instance of a ctypes type. *offset* defaults to zero, and must be an integer that will be added to the internal pointer value.

`byref(obj, offset)` corresponds to this C code :

```
(((char *)&obj) + offset)
```

The returned object can only be used as a foreign function call parameter. It behaves similar to `pointer(obj)`, but the construction is a lot faster.

`ctypes.`**`CopyComPointer`**(*src*, *dst*)

Copies a COM pointer from *src* to *dst* and returns the Windows specific `HRESULT` value.

If *src* is not `NULL`, its `AddRef` method is called, incrementing the reference count.

In contrast, the reference count of *dst* will not be decremented before assigning the new value. Unless *dst* is `NULL`, the caller is responsible for decrementing the reference count by calling its `Release` method when necessary.

Availability : Windows

Ajouté dans la version 3.14.

`ctypes.`**`cast`**(*obj*, *type*)

This function is similar to the cast operator in C. It returns a new instance of *type* which points to the same memory block as *obj*. *type* must be a pointer type, and *obj* must be an object that can be interpreted as a pointer.

`ctypes.`**`create_string_buffer`**(*init_or_size*, *size=None*)

This function creates a mutable character buffer. The returned object is a ctypes array of *c_char*.

init_or_size must be an integer which specifies the size of the array, or a bytes object which will be used to initialize the array items.

If a bytes object is specified as first argument, the buffer is made one item larger than its length so that the last element in the array is a NUL termination character. An integer can be passed as second argument which allows specifying the size of the array if the length of the bytes should not be used.

Raises an *auditing event* `ctypes.create_string_buffer` with arguments `init`, `size`.

`ctypes.create_unicode_buffer(init_or_size, size=None)`

This function creates a mutable unicode character buffer. The returned object is a ctypes array of `c_wchar`.

init_or_size must be an integer which specifies the size of the array, or a string which will be used to initialize the array items.

If a string is specified as first argument, the buffer is made one item larger than the length of the string so that the last element in the array is a NUL termination character. An integer can be passed as second argument which allows specifying the size of the array if the length of the string should not be used.

Raises an *auditing event* `ctypes.create_unicode_buffer` with arguments `init`, `size`.

`ctypes.DllCanUnloadNow()`

This function is a hook which allows implementing in-process COM servers with ctypes. It is called from the DllCanUnloadNow function that the _ctypes extension dll exports.

Availability : Windows

`ctypes.DllGetClassObject()`

This function is a hook which allows implementing in-process COM servers with ctypes. It is called from the DllGetClassObject function that the `_ctypes` extension dll exports.

Availability : Windows

`ctypes.util.find_library(name)`

Try to find a library and return a pathname. *name* is the library name without any prefix like `lib`, suffix like `.so`, `.dylib` or version number (this is the form used for the posix linker option `-l`). If no library can be found, returns `None`.

Le mode opératoire exact dépend du système.

`ctypes.util.find_msvcrt()`

Returns the filename of the VC runtime library used by Python, and by the extension modules. If the name of the library cannot be determined, `None` is returned.

If you need to free memory, for example, allocated by an extension module with a call to the `free(void *)`, it is important that you use the function in the same library that allocated the memory.

Availability : Windows

`ctypes.util.dllist()`

Try to provide a list of paths of the shared libraries loaded into the current process. These paths are not normalized or processed in any way. The function can raise `OSError` if the underlying platform APIs fail. The exact functionality is system dependent.

On most platforms, the first element of the list represents the current executable file. It may be an empty string.

Availability : Windows, macOS, iOS, glibc, BSD libc, musl

Ajouté dans la version 3.14.

`ctypes.FormatError([code])`

Returns a textual description of the error code *code*. If no error code is specified, the last error code is used by calling the Windows API function `GetLastError()`.

Availability : Windows

`ctypes.GetLastError()`

Returns the last error code set by Windows in the calling thread. This function calls the Windows `GetLastError()` function directly, it does not return the ctypes-private copy of the error code.

Availability : Windows

`ctypes.get_errno()`

Returns the current value of the ctypes-private copy of the system `errno` variable in the calling thread.

Raises an *auditing event* `ctypes.get_errno` with no arguments.

`ctypes.get_last_error()`

Returns the current value of the ctypes-private copy of the system `LastError` variable in the calling thread.

Availability : Windows

Raises an *auditing event* `ctypes.get_last_error` with no arguments.

`ctypes.memmove(dst, src, count)`

Same as the standard C memmove library function : copies *count* bytes from *src* to *dst*. *dst* and *src* must be integers or ctypes instances that can be converted to pointers.

`ctypes.memset(dst, c, count)`

Same as the standard C memset library function : fills the memory block at address *dst* with *count* bytes of value *c*. *dst* must be an integer specifying an address, or a ctypes instance.

`ctypes.POINTER(type, /)`

Create and return a new ctypes pointer type. Pointer types are cached and reused internally, so calling this function repeatedly is cheap. *type* must be a ctypes type.

`ctypes.pointer(obj, /)`

Create a new pointer instance, pointing to *obj*. The returned object is of the type `POINTER(type(obj))`.

Note : If you just want to pass a pointer to an object to a foreign function call, you should use `byref(obj)` which is much faster.

`ctypes.resize(obj, size)`

This function resizes the internal memory buffer of *obj*, which must be an instance of a ctypes type. It is not possible to make the buffer smaller than the native size of the objects type, as given by `sizeof(type(obj))`, but it is possible to enlarge the buffer.

`ctypes.set_errno(value)`

Set the current value of the ctypes-private copy of the system `errno` variable in the calling thread to *value* and return the previous value.

Raises an *auditing event* `ctypes.set_errno` with argument `errno`.

`ctypes.set_last_error(value)`

Sets the current value of the ctypes-private copy of the system `LastError` variable in the calling thread to *value* and return the previous value.

Availability : Windows

Raises an *auditing event* `ctypes.set_last_error` with argument `error`.

`ctypes.sizeof(obj_or_type)`

Returns the size in bytes of a ctypes type or instance memory buffer. Does the same as the C `sizeof` operator.

`ctypes.string_at(ptr, size=-1)`

Return the byte string at *void *ptr*. If *size* is specified, it is used as size, otherwise the string is assumed to be zero-terminated.

Raises an *auditing event* `ctypes.string_at` with arguments `ptr`, `size`.

`ctypes.WinError(code=None, descr=None)`

Creates an instance of `OSError`. If *code* is not specified, `GetLastError()` is called to determine the error code. If *descr* is not specified, `FormatError()` is called to get a textual description of the error.

Availability : Windows

Modifié dans la version 3.3 : An instance of `WindowsError` used to be created, which is now an alias of `OSError`.

`ctypes.wstring_at(ptr, size=-1)`

Return the wide-character string at *void *ptr*. If *size* is specified, it is used as the number of characters of the string, otherwise the string is assumed to be zero-terminated.

Raises an *auditing event* `ctypes.wstring_at` with arguments `ptr`, `size`.

`ctypes.memoryview_at`(*ptr*, *size*, *readonly=False*)

Return a `memoryview` object of length *size* that references memory starting at *void *ptr*.

If *readonly* is true, the returned `memoryview` object can not be used to modify the underlying memory. (Changes made by other means will still be reflected in the returned object.)

This function is similar to `string_at()` with the key difference of not making a copy of the specified memory. It is a semantically equivalent (but more efficient) alternative to `memoryview((c_byte * size).from_address(ptr))`. (While `from_address()` only takes integers, *ptr* can also be given as a `ctypes.POINTER` or a `byref()` object.)

Raises an *auditing event* `ctypes.memoryview_at` with arguments `address`, `size`, `readonly`.

Ajouté dans la version 3.14.

Types de données

class `ctypes._CData`

This non-public class is the common base class of all ctypes data types. Among other things, all ctypes type instances contain a memory block that hold C compatible data ; the address of the memory block is returned by the `addressof()` helper function. Another instance variable is exposed as `_objects` ; this contains other Python objects that need to be kept alive in case the memory block contains pointers.

Common methods of ctypes data types, these are all class methods (to be exact, they are methods of the *metaclass*) :

from_buffer(*source*[, *offset*])

This method returns a ctypes instance that shares the buffer of the *source* object. The *source* object must support the writeable buffer interface. The optional *offset* parameter specifies an offset into the source buffer in bytes ; the default is zero. If the source buffer is not large enough a `ValueError` is raised.

Raises an *auditing event* `ctypes.cdata/buffer` with arguments `pointer`, `size`, `offset`.

from_buffer_copy(*source*[, *offset*])

This method creates a ctypes instance, copying the buffer from the *source* object buffer which must be readable. The optional *offset* parameter specifies an offset into the source buffer in bytes ; the default is zero. If the source buffer is not large enough a `ValueError` is raised.

Raises an *auditing event* `ctypes.cdata/buffer` with arguments `pointer`, `size`, `offset`.

from_address(*address*)

This method returns a ctypes type instance using the memory specified by *address* which must be an integer.

This method, and others that indirectly call this method, raises an *auditing event* `ctypes.cdata` with argument `address`.

from_param(*obj*)

This method adapts *obj* to a ctypes type. It is called with the actual object used in a foreign function call when the type is present in the foreign function's `argtypes` tuple ; it must return an object that can be used as a function call parameter.

All ctypes data types have a default implementation of this classmethod that normally returns *obj* if that is an instance of the type. Some types accept other objects as well.

in_dll(*library*, *name*)

This method returns a ctypes type instance exported by a shared library. *name* is the name of the symbol that exports the data, *library* is the loaded shared library.

Common instance variables of ctypes data types :

_b_base_

Sometimes ctypes data instances do not own the memory block they contain, instead they share part of the memory block of a base object. The *_b_base_* read-only member is the root ctypes object that owns the memory block.

_b_needsfree_

This read-only variable is true when the ctypes data instance has allocated the memory block itself, false otherwise.

_objects

This member is either `None` or a dictionary containing Python objects that need to be kept alive so that the memory block contents is kept valid. This object is only exposed for debugging ; never modify the contents of this dictionary.

Types de données de base

class ctypes.**_SimpleCData**

This non-public class is the base class of all fundamental ctypes data types. It is mentioned here because it contains the common attributes of the fundamental ctypes data types. *_SimpleCData* is a subclass of *_CData*, so it inherits their methods and attributes. ctypes data types that are not and do not contain pointers can now be pickled.

Instances have a single attribute :

value

This attribute contains the actual value of the instance. For integer and pointer types, it is an integer, for character types, it is a single character bytes object or string, for character pointer types it is a Python bytes object or string.

When the `value` attribute is retrieved from a ctypes instance, usually a new object is returned each time. *ctypes* does *not* implement original object return, always a new object is constructed. The same is true for all other ctypes object instances.

Fundamental data types, when returned as foreign function call results, or, for example, by retrieving structure field members or array items, are transparently converted to native Python types. In other words, if a foreign function has a *restype* of *c_char_p*, you will always receive a Python bytes object, *not* a *c_char_p* instance.

Subclasses of fundamental data types do *not* inherit this behavior. So, if a foreign functions `restype` is a subclass of *c_void_p*, you will receive an instance of this subclass from the function call. Of course, you can get the value of the pointer by accessing the `value` attribute.

These are the fundamental ctypes data types :

class ctypes.**c_byte**

Represents the C `signed char` datatype, and interprets the value as small integer. The constructor accepts an optional integer initializer ; no overflow checking is done.

class ctypes.**c_char**

Represents the C `char` datatype, and interprets the value as a single character. The constructor accepts an optional string initializer, the length of the string must be exactly one character.

class ctypes.**c_char_p**

Represents the C `char*` datatype when it points to a zero-terminated string. For a general character pointer that may also point to binary data, `POINTER(c_char)` must be used. The constructor accepts an integer address, or a bytes object.

class ctypes.**c_double**

Represents the C `double` datatype. The constructor accepts an optional float initializer.

class ctypes.**c_longdouble**

Represents the C `long double` datatype. The constructor accepts an optional float initializer. On platforms where `sizeof(long double) == sizeof(double)` it is an alias to *c_double*.

class ctypes.**c_float**

Represents the C `float` datatype. The constructor accepts an optional float initializer.

class ctypes.**c_double_complex**

Represents the C `double complex` datatype, if available. The constructor accepts an optional *complex* initializer.

Ajouté dans la version 3.14.

class ctypes.**c_float_complex**

Represents the C `float complex` datatype, if available. The constructor accepts an optional *complex* initializer.

Ajouté dans la version 3.14.

class ctypes.**c_longdouble_complex**

Represents the C `long double complex` datatype, if available. The constructor accepts an optional *complex* initializer.

Ajouté dans la version 3.14.

class ctypes.**c_int**

Represents the C `signed int` datatype. The constructor accepts an optional integer initializer ; no overflow checking is done. On platforms where `sizeof(int) == sizeof(long)` it is an alias to *c_long*.

class ctypes.**c_int8**

Represents the C 8-bit `signed int` datatype. Usually an alias for *c_byte*.

class ctypes.**c_int16**

Represents the C 16-bit `signed int` datatype. Usually an alias for *c_short*.

class ctypes.**c_int32**

Represents the C 32-bit `signed int` datatype. Usually an alias for *c_int*.

class ctypes.**c_int64**

Represents the C 64-bit `signed int` datatype. Usually an alias for *c_longlong*.

class ctypes.**c_long**

Represents the C `signed long` datatype. The constructor accepts an optional integer initializer ; no overflow checking is done.

class ctypes.**c_longlong**

Represents the C `signed long long` datatype. The constructor accepts an optional integer initializer ; no overflow checking is done.

class ctypes.**c_short**

Represents the C `signed short` datatype. The constructor accepts an optional integer initializer ; no overflow checking is done.

class ctypes.**c_size_t**

Represents the C `size_t` datatype.

class ctypes.**c_ssize_t**

Represents the C `ssize_t` datatype.

Ajouté dans la version 3.2.

class ctypes.**c_time_t**

Represents the C `time_t` datatype.

Ajouté dans la version 3.12.

class ctypes.**c_ubyte**

Represents the C `unsigned char` datatype, it interprets the value as small integer. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.**c_uint**

Represents the C `unsigned int` datatype. The constructor accepts an optional integer initializer; no overflow checking is done. On platforms where `sizeof(int) == sizeof(long)` it is an alias for *c_ulong*.

class ctypes.**c_uint8**

Represents the C 8-bit `unsigned int` datatype. Usually an alias for *c_ubyte*.

class ctypes.**c_uint16**

Represents the C 16-bit `unsigned int` datatype. Usually an alias for *c_ushort*.

class ctypes.**c_uint32**

Represents the C 32-bit `unsigned int` datatype. Usually an alias for *c_uint*.

class ctypes.**c_uint64**

Represents the C 64-bit `unsigned int` datatype. Usually an alias for *c_ulonglong*.

class ctypes.**c_ulong**

Represents the C `unsigned long` datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.**c_ulonglong**

Represents the C `unsigned long long` datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.**c_ushort**

Represents the C `unsigned short` datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.**c_void_p**

Represents the C `void*` type. The value is represented as integer. The constructor accepts an optional integer initializer.

class ctypes.**c_wchar**

Represents the C `wchar_t` datatype, and interprets the value as a single character unicode string. The constructor accepts an optional string initializer, the length of the string must be exactly one character.

class ctypes.**c_wchar_p**

Represents the C `wchar_t*` datatype, which must be a pointer to a zero-terminated wide character string. The constructor accepts an integer address, or a string.

class ctypes.**c_bool**

Represent the C `bool` datatype (more accurately, `_Bool` from C99). Its value can be `True` or `False`, and the constructor accepts any object that has a truth value.

class ctypes.**HRESULT**

Represents a `HRESULT` value, which contains success or error information for a function or method call.

Availability : Windows

class ctypes.**py_object**

Represents the C `PyObject*` datatype. Calling this without an argument creates a `NULL` `PyObject*` pointer.

The `ctypes.wintypes` module provides quite some other Windows specific data types, for example `HWND`, `WPARAM`, or `DWORD`. Some useful structures like `MSG` or `RECT` are also defined.

Types de données dérivés de Structure

class ctypes.**Union**(**args*, ***kw*)

Abstract base class for unions in native byte order.

Unions share common attributes and behavior with structures ; see `Structure` documentation for details.

class ctypes.**BigEndianUnion**(**args*, ***kw*)

Abstract base class for unions in *big endian* byte order.

Ajouté dans la version 3.11.

class ctypes.**LittleEndianUnion**(**args*, ***kw*)

Abstract base class for unions in *little endian* byte order.

Ajouté dans la version 3.11.

class ctypes.**BigEndianStructure**(**args*, ***kw*)

Abstract base class for structures in *big endian* byte order.

class ctypes.**LittleEndianStructure**(**args*, ***kw*)

Abstract base class for structures in *little endian* byte order.

Structures and unions with non-native byte order cannot contain pointer type fields, or any other data types containing pointer type fields.

class ctypes.**Structure**(**args*, ***kw*)

Abstract base class for structures in *native* byte order.

Concrete structure and union types must be created by subclassing one of these types, and at least define a `_fields_` class variable. `ctypes` will create *descriptor*s which allow reading and writing the fields by direct attribute accesses. These are the

fields

A sequence defining the structure fields. The items must be 2-tuples or 3-tuples. The first item is the name of the field, the second item specifies the type of the field ; it can be any ctypes data type.

For integer type fields like `c_int`, a third optional item can be given. It must be a small positive integer defining the bit width of the field.

Field names must be unique within one structure or union. This is not checked, only one field can be accessed when names are repeated.

It is possible to define the `_fields_` class variable *after* the class statement that defines the Structure subclass, this allows creating data types that directly or indirectly reference themselves :

```
class List(Structure):
    pass
List._fields_ = [("pnext", POINTER(List)),
                 ...
                ]
```

The `_fields_` class variable can only be set once. Later assignments will raise an `AttributeError`.

Additionally, the `_fields_` class variable must be defined before the structure or union type is first used : an instance or subclass is created, `sizeof()` is called on it, and so on. Later assignments to `_fields_` will raise an `AttributeError`. If `_fields_` has not been set before such use, the structure or union will have no own fields, as if `_fields_` was empty.

Sub-subclasses of structure types inherit the fields of the base class plus the `_fields_` defined in the sub-subclass, if any.

pack

An optional small integer that allows overriding the alignment of structure fields in the instance. `_pack_` must already be defined when `_fields_` is assigned, otherwise it will have no effect. Setting this attribute to 0 is the same as not setting it at all.

align

An optional small integer that allows overriding the alignment of the structure when being packed or unpacked to/from memory. Setting this attribute to 0 is the same as not setting it at all.

Ajouté dans la version 3.13.

layout

An optional string naming the struct/union layout. It can currently be set to :

— `"ms"` : the layout used by the Microsoft compiler (MSVC). On GCC and Clang, this layout can be selected with `__attribute__((ms_struct))`.

— `"gcc-sysv"` : the layout used by GCC with the System V or "SysV-like" data model, as used on Linux and macOS. With this layout, *_pack_* must be unset or zero.

If not set explicitly, `ctypes` will use a default that matches the platform conventions. This default may change in future Python releases (for example, when a new platform gains official support, or when a difference between similar platforms is found). Currently the default will be :

— On Windows : `"ms"`

— When *_pack_* is specified : `"ms"`

— Otherwise : `"gcc-sysv"`

`_layout_` must already be defined when *_fields_* is assigned, otherwise it will have no effect.

anonymous

An optional sequence that lists the names of unnamed (anonymous) fields. *_anonymous_* must be already defined when *_fields_* is assigned, otherwise it will have no effect.

The fields listed in this variable must be structure or union type fields. *ctypes* will create descriptors in the structure type that allows accessing the nested fields directly, without the need to create the structure or union field.

Here is an example type (Windows) :

```
class _U(Union):
    _fields_ = [("lptdesc", POINTER(TYPEDESC)),
                ("lpadesc", POINTER(ARRAYDESC)),
                ("hreftype", HREFTYPE)]

class TYPEDESC(Structure):
    _anonymous_ = ("u",)
    _fields_ = [("u", _U),
                ("vt", VARTYPE)]
```

The `TYPEDESC` structure describes a COM data type, the `vt` field specifies which one of the union fields is valid. Since the `u` field is defined as anonymous field, it is now possible to access the members directly off the TYPEDESC instance. `td.lptdesc` and `td.u.lptdesc` are equivalent, but the former is faster since it does not need to create a temporary union instance :

```
td = TYPEDESC()
td.vt = VT_PTR
td.lptdesc = POINTER(some_type)
td.u.lptdesc = POINTER(some_type)
```

It is possible to define sub-subclasses of structures, they inherit the fields of the base class. If the subclass definition has a separate *_fields_* variable, the fields specified in this are appended to the fields of the base class.

Structure and union constructors accept both positional and keyword arguments. Positional arguments are used to initialize member fields in the same order as they are appear in *_fields_*. Keyword arguments in the constructor are interpreted as attribute assignments, so they will initialize *_fields_* with the same name, or create new attributes for names not present in *_fields_*.

Tableaux et pointeurs

class ctypes.**Array**(**args*)

Classe de base abstraite pour les *arrays*.

The recommended way to create concrete array types is by multiplying any `ctypes` data type with a non-negative integer. Alternatively, you can subclass this type and define `_length_` and `_type_` class variables. Array elements can be read and written using standard subscript and slice accesses ; for slice reads, the resulting object is *not* itself an `Array`.

length

A positive integer specifying the number of elements in the array. Out-of-range subscripts result in an `IndexError`. Will be returned by `len()`.

type

Spécifie le type de chaque élément de l'*array*.

Array subclass constructors accept positional arguments, used to initialize the elements in order.

ctypes.**ARRAY**(*type*, *length*)

Create an array. Equivalent to `type * length`, where *type* is a `ctypes` data type and *length* an integer.

This function is *soft deprecated* in favor of multiplication. There are no plans to remove it.

class ctypes.**_Pointer**

Private, abstract base class for pointers.

Concrete pointer types are created by calling `POINTER()` with the type that will be pointed to ; this is done automatically by `pointer()`.

If a pointer points to an array, its elements can be read and written using standard subscript and slice accesses. Pointer objects have no size, so `len()` will raise `TypeError`. Negative subscripts will read from the memory *before* the pointer (as in C), and out-of-range subscripts will probably crash with an access violation (if you're lucky).

type

Specifies the type pointed to.

contents

Returns the object to which to pointer points. Assigning to this attribute changes the pointer to point to the assigned object.

Exceptions

exception ctypes.**ArgumentError**

Exception levée quand un appel à la fonction externe ne peut pas convertir un des arguments qu'elle a reçus.

exception ctypes.**COMError**(*hresult*, *text*, *details*)

This exception is raised when a COM method call failed.

hresult

The integer value representing the error code.

text

The error message.

details

The 5-tuple `(descr, source, helpfile, helpcontext, progid)`.

descr is the textual description. *source* is the language-dependent `ProgID` for the class or application that raised the error. *helpfile* is the path of the help file. *helpcontext* is the help context identifier. *progid* is the `ProgID` of the interface that defined the error.

Availability : Windows

Ajouté dans la version 3.14.

CHAPITRE 17

Command Line Interface Libraries

The modules described in this chapter assist with implementing command line and terminal interfaces for applications.

Here's an overview :

17.1 `argparse` --- Parser for command-line options, arguments and subcommands

Ajouté dans la version 3.2.

Code source : Lib/argparse.py

> **Note**
>
> While `argparse` is the default recommended standard library module for implementing basic command line applications, authors with more exacting requirements for exactly how their command line applications behave may find it doesn't provide the necessary level of control. Refer to *Choosing an argument parsing library* for alternatives to consider when `argparse` doesn't support behaviors that the application requires (such as entirely disabling support for interspersed options and positional arguments, or accepting option parameter values that start with – even when they correspond to another defined option).

> **Tutoriel**
>
> Cette page est la référence de l'API. Pour une introduction plus en douceur à l'analyse des arguments de la ligne de commande, regardez *le tutoriel argparse*.

The `argparse` module makes it easy to write user-friendly command-line interfaces. The program defines what arguments it requires, and `argparse` will figure out how to parse those out of `sys.argv`. The `argparse` module also automatically generates help and usage messages. The module will also issue errors when users give the program invalid arguments.

The `argparse` module's support for command-line interfaces is built around an instance of `argparse.ArgumentParser`. It is a container for argument specifications and has options that apply to the parser as whole :

```
parser = argparse.ArgumentParser(
                    prog='ProgramName',
                    description='What the program does',
                    epilog='Text at the bottom of help')
```

La méthode `ArgumentParser.add_argument()` permet de définir les arguments de l'analyseur. Ceux-ci peuvent être des arguments positionnels, des arguments optionnels ou des drapeaux (qui sont alors traduits en valeurs booléennes). Les arguments ont la possibilité d'être complétés par des valeurs :

```
parser.add_argument('filename')           # positional argument
parser.add_argument('-c', '--count')      # option that takes a value
parser.add_argument('-v', '--verbose',
                    action='store_true')  # on/off flag
```

La méthode `ArgumentParser.parse_args()` lance l'analyseur et stocke les résultats dans un objet `argparse.Namespace` :

```
args = parser.parse_args()
print(args.filename, args.count, args.verbose)
```

Note

If you're looking for a guide about how to upgrade `optparse` code to `argparse`, see *Upgrading Optparse Code*.

17.1.1 Objets `ArgumentParser`

class argparse.**ArgumentParser**(*prog=None*, *usage=None*, *description=None*, *epilog=None*, *parents=[]*, *formatter_class=argparse.HelpFormatter*, *prefix_chars='-'*, *fromfile_prefix_chars=None*, *argument_default=None*, *conflict_handler='error'*, *add_help=True*, *allow_abbrev=True*, *exit_on_error=True*, *suggest_on_error=False*)

Crée un nouvel objet `ArgumentParser`. Tous les paramètres doivent être passés en arguments nommés. Chaque paramètre a sa propre description détaillée ci-dessous, mais en résumé ils sont :

- *prog* - The name of the program (default : generated from the `__main__` module attributes and `sys.argv[0]`)
- *usage* – Chaîne décrivant l'utilisation du programme (par défaut : générée à partir des arguments ajoutés à l'analyseur) ;
- *description* – Texte à afficher au dessus de l'aide pour les arguments (vide par défaut) ;
- *epilog* – Texte à afficher après l'aide des arguments (vide par défaut) ;
- *parents* – Liste d'objets `ArgumentParser` contenant des arguments qui devraient aussi être inclus ;
- *formatter_class* – Classe pour personnaliser la sortie du message d'aide ;
- *prefix_chars* – Jeu de caractères qui précède les arguments optionnels (par défaut : `'-'`) ;
- *fromfile_prefix_chars* – Jeu de caractères qui précède les fichiers d'où des arguments additionnels doivent être lus (par défaut : `None`) ;
- *argument_default* – Valeur globale par défaut pour les arguments (par défaut : `None`) ;
- *conflict_handler* – Stratégie pour résoudre les conflits entre les arguments optionnels (non-nécessaire en général) ;
- *add_help* – Ajoute une option d'aide `-h/--help` à l'analyseur (par défaut : `True`) ;
- *allow_abbrev* – Permet l'acceptation d'abréviations non-ambigües pour les options longues (par défaut : `True`) ;
- *exit_on_error* - Determines whether or not `ArgumentParser` exits with error info when an error occurs. (default : `True`)
- *suggest_on_error* - Enables suggestions for mistyped argument choices and subparser names (default : `False`)

Modifié dans la version 3.5 : Le paramètre *allow_abbrev* est ajouté.

Modifié dans la version 3.8 : Dans les versions précédentes, *allow_abbrev* désactivait aussi le regroupement de plusieurs options courtes telles que `-vv` pour signifier `-v -v`.

Modifié dans la version 3.9 : Le paramètre *exit_on_error* est ajouté.

Les sections suivantes décrivent comment chacune de ces options sont utilisées.

Le paramètre *prog*

By default, `ArgumentParser` calculates the name of the program to display in help messages depending on the way the Python interpreter was run :

— The `base name` of `sys.argv[0]` if a file was passed as argument.
— The Python interpreter name followed by `sys.argv[0]` if a directory or a zipfile was passed as argument.
— The Python interpreter name followed by `-m` followed by the module or package name if the `-m` option was used.

This default is almost always desirable because it will make the help messages match the string that was used to invoke the program on the command line. However, to change this default behavior, another value can be supplied using the `prog=` argument to `ArgumentParser` :

```
>>> parser = argparse.ArgumentParser(prog='myprogram')
>>> parser.print_help()
usage: myprogram [-h]

options:
 -h, --help  show this help message and exit
```

Note that the program name, whether determined from `sys.argv[0]`, from the `__main__` module attributes or from the `prog=` argument, is available to help messages using the `%(prog)s` format specifier.

```
>>> parser = argparse.ArgumentParser(prog='myprogram')
>>> parser.add_argument('--foo', help='foo of the %(prog)s program')
>>> parser.print_help()
usage: myprogram [-h] [--foo FOO]

options:
 -h, --help  show this help message and exit
 --foo FOO   foo of the myprogram program
```

Modifié dans la version 3.14 : The default `prog` value now reflects how `__main__` was actually executed, rather than always being `os.path.basename(sys.argv[0])`.

Le paramètre *usage*

By default, `ArgumentParser` calculates the usage message from the arguments it contains. The default message can be overridden with the `usage=` keyword argument :

```
>>> parser = argparse.ArgumentParser(prog='PROG', usage='%(prog)s [options]')
>>> parser.add_argument('--foo', nargs='?', help='foo help')
>>> parser.add_argument('bar', nargs='+', help='bar help')
>>> parser.print_help()
usage: PROG [options]

positional arguments:
 bar          bar help

options:
 -h, --help   show this help message and exit
 --foo [FOO]  foo help
```

Le spécificateur de formatage `%(prog)s` est disponible pour insérer le nom du programme dans vos messages d'utilisation.

When a custom usage message is specified for the main parser, you may also want to consider passing the `prog` argument to *add_subparsers()* or the `prog` and the `usage` arguments to `add_parser()`, to ensure consistent command prefixes and usage information across subparsers.

Le paramètre *description*

Most calls to the *ArgumentParser* constructor will use the `description=` keyword argument. This argument gives a brief description of what the program does and how it works. In help messages, the description is displayed between the command-line usage string and the help messages for the various arguments.

Par défaut, la description est sujette au retour à la ligne automatique pour se conformer à l'espace disponible. Pour changer ce comportement, voyez l'argument *formatter_class*.

Le paramètre *epilog*

Certains programmes aiment afficher un texte supplémentaire après la description des arguments. Un tel texte peut être spécifié grâce à l'argument `epilog=` du constructeur d'*ArgumentParser* :

```
>>> parser = argparse.ArgumentParser(
...     description='A foo that bars',
...     epilog="And that's how you'd foo a bar")
>>> parser.print_help()
usage: argparse.py [-h]

A foo that bars

options:
 -h, --help  show this help message and exit

And that's how you'd foo a bar
```

De même que pour l'argument *description*, le texte passé à `epilog=` est sujet au retour à la ligne automatique. Ce comportement peut être ajusté grâce à l'argument *formatter_class* du constructeur d'*ArgumentParser*.

Le paramètre *parents*

Parfois, plusieurs analyseurs partagent un jeu commun d'arguments. Plutôt que de répéter les définitions de ces arguments, un analyseur commun qui contient tous les arguments partagés peut être utilisé, puis passé à l'argument `parents=` du constructeur d'*ArgumentParser*. L'argument `parents=` accepte une liste d'objets *ArgumentParser*, accumule toutes les actions positionnelles et optionnelles de ces objets, puis les ajoute à l'instance d'*ArgumentParser* en cours de création :

```
>>> parent_parser = argparse.ArgumentParser(add_help=False)
>>> parent_parser.add_argument('--parent', type=int)

>>> foo_parser = argparse.ArgumentParser(parents=[parent_parser])
>>> foo_parser.add_argument('foo')
>>> foo_parser.parse_args(['--parent', '2', 'XXX'])
Namespace(foo='XXX', parent=2)

>>> bar_parser = argparse.ArgumentParser(parents=[parent_parser])
>>> bar_parser.add_argument('--bar')
>>> bar_parser.parse_args(['--bar', 'YYY'])
Namespace(bar='YYY', parent=None)
```

Prenez note que la majorité des analyseurs parents doivent définir `add_help=False`. Autrement, le constructeur d'*ArgumentParser* va voir plus d'une option `-h/--help` (une pour le parent et une pour l'instance en cours de

création) et va lever une erreur.

> **Note**
>
> Vous devez initialiser complètement les analyseurs avant de les passer à `parents=`. Si vous changez les analyseurs parents après la création de l'analyseur enfant, ces changements ne sont pas répercutés sur l'enfant.

Le paramètre *formatter_class*

Les objets `ArgumentParser` permettent la personnalisation de la mise en page des messages d'aide en spécifiant une classe de formatage alternative. Il y a actuellement quatre classes de formatage :

class argparse.**RawDescriptionHelpFormatter**

class argparse.**RawTextHelpFormatter**

class argparse.**ArgumentDefaultsHelpFormatter**

class argparse.**MetavarTypeHelpFormatter**

`RawDescriptionHelpFormatter` et `RawTextHelpFormatter` vous donnent plus de contrôle sur comment les descriptions textuelles sont affichées. Par défaut, les contenus de *description* et *epilog* des objets `ArgumentParser` font l'objet du retour à la ligne automatique dans les messages d'aide :

```
>>> parser = argparse.ArgumentParser(
...     prog='PROG',
...     description='''this description
...         was indented weird
...             but that is okay''',
...     epilog='''
...             likewise for this epilog whose whitespace will
...         be cleaned up and whose words will be wrapped
...         across a couple lines''')
>>> parser.print_help()
usage: PROG [-h]

this description was indented weird but that is okay

options:
 -h, --help  show this help message and exit

likewise for this epilog whose whitespace will be cleaned up and whose words
will be wrapped across a couple lines
```

Passer `RawDescriptionHelpFormatter` à `formatter_class=` indique que les textes de *description* et d'*epilog* ont déjà été formatés correctement et qu'ils ne doivent pas faire l'objet d'un retour à la ligne automatique :

```
>>> parser = argparse.ArgumentParser(
...     prog='PROG',
...     formatter_class=argparse.RawDescriptionHelpFormatter,
...     description=textwrap.dedent('''\
...         Please do not mess up this text!
...         --------------------------------
...             I have indented it
...             exactly the way
...             I want it
...         '''))
>>> parser.print_help()
usage: PROG [-h]
```

(suite sur la page suivante)

(suite de la page précédente)

```
Please do not mess up this text!
--------------------------------
   I have indented it
   exactly the way
   I want it

options:
 -h, --help  show this help message and exit
```

`RawTextHelpFormatter` maintains whitespace for all sorts of help text, including argument descriptions. However, multiple newlines are replaced with one. If you wish to preserve multiple blank lines, add spaces between the newlines.

`ArgumentDefaultsHelpFormatter` ajoute automatiquement l'information sur les valeurs par défaut aux messages d'aide de tous les arguments :

```
>>> parser = argparse.ArgumentParser(
...     prog='PROG',
...     formatter_class=argparse.ArgumentDefaultsHelpFormatter)
>>> parser.add_argument('--foo', type=int, default=42, help='FOO!')
>>> parser.add_argument('bar', nargs='*', default=[1, 2, 3], help='BAR!')
>>> parser.print_help()
usage: PROG [-h] [--foo FOO] [bar ...]

positional arguments:
 bar         BAR! (default: [1, 2, 3])

options:
 -h, --help  show this help message and exit
 --foo FOO   FOO! (default: 42)
```

`MetavarTypeHelpFormatter` utilise le nom du *type* de l'argument pour chacun des arguments comme nom d'affichage pour leurs valeurs (contrairement au formateur standard qui utilise *dest*) :

```
>>> parser = argparse.ArgumentParser(
...     prog='PROG',
...     formatter_class=argparse.MetavarTypeHelpFormatter)
>>> parser.add_argument('--foo', type=int)
>>> parser.add_argument('bar', type=float)
>>> parser.print_help()
usage: PROG [-h] [--foo int] float

positional arguments:
  float

options:
  -h, --help  show this help message and exit
  --foo int
```

Le paramètre *prefix_chars*

Most command-line options will use – as the prefix, e.g. `-f/--foo`. Parsers that need to support different or additional prefix characters, e.g. for options like `+f` or `/foo`, may specify them using the `prefix_chars=` argument to the `ArgumentParser` constructor :

```
>>> parser = argparse.ArgumentParser(prog='PROG', prefix_chars='-+')
>>> parser.add_argument('+f')
>>> parser.add_argument('++bar')
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> parser.parse_args('+f X ++bar Y'.split())
Namespace(bar='Y', f='X')
```

La valeur par défaut de `prefix_chars=` est `'-'`. Passer un jeu de caractères qui n'inclut pas `-` provoquera le refus des options comme `-f/--foo`.

Le paramètre *fromfile_prefix_chars*

Parfois, par exemple quand on traite une liste d'arguments particulièrement longue, il est logique de stocker la liste d'arguments dans un fichier plutôt que de la saisir sur la ligne de commande. Si un jeu de caractères est passé à l'argument `fromfile_prefix_chars=` du constructeur de `ArgumentParser`, alors les arguments qui commencent par l'un des caractères spécifiés sont traités comme des fichiers et sont remplacés par les arguments contenus dans ces fichiers. Par exemple :

```
>>> with open('args.txt', 'w', encoding=sys.getfilesystemencoding()) as fp:
...     fp.write('-f\nbar')
...
>>> parser = argparse.ArgumentParser(fromfile_prefix_chars='@')
>>> parser.add_argument('-f')
>>> parser.parse_args(['-f', 'foo', '@args.txt'])
Namespace(f='bar')
```

Par défaut, les arguments lus à partir d'un fichier doivent être chacun sur une nouvelle ligne (voir aussi `convert_arg_line_to_args()`) et ils sont traités comme s'ils étaient au même emplacement que le fichier original référençant les arguments de la ligne de commande. Ainsi dans l'exemple ci-dessus, l'expression `['-f', 'foo', '@args.txt']` est équivalente à l'expression `['-f', 'foo', '-f', 'bar']`.

`ArgumentParser` uses *filesystem encoding and error handler* to read the file containing arguments.

Par défaut, l'argument `fromfile_prefix_chars=` est `None`, ce qui signifie que les arguments ne sont pas traités en tant que références à des fichiers.

Modifié dans la version 3.12 : `ArgumentParser` changed encoding and errors to read arguments files from default (e.g. `locale.getpreferredencoding(False)` and `"strict"`) to the *filesystem encoding and error handler*. Arguments file should be encoded in UTF-8 instead of ANSI Codepage on Windows.

Le paramètre *argument_default*

Généralement, les valeurs par défaut des arguments sont définies soit en passant la valeur désirée à `add_argument()` soit par un appel à la méthode `set_defaults()`. Cette méthode accepte un ensemble de paires nom-valeur. Il est parfois pertinent de configurer une valeur par défaut pour tous les arguments d'un analyseur. On peut activer ce comportement en passant la valeur désirée à l'argument nommé `argument_default=` du constructeur de `ArgumentParser`. Par exemple, pour supprimer globalement la création d'attributs pendant l'appel de `parse_args()`, on fournit `argument_default=SUPPRESS` :

```
>>> parser = argparse.ArgumentParser(argument_default=argparse.SUPPRESS)
>>> parser.add_argument('--foo')
>>> parser.add_argument('bar', nargs='?')
>>> parser.parse_args(['--foo', '1', 'BAR'])
Namespace(bar='BAR', foo='1')
>>> parser.parse_args([])
Namespace()
```

Le paramètre *allow_abbrev*

En temps normal, lorsque vous passez une liste d'arguments à la méthode `parse_args()` d'`ArgumentParser` *elle accepte les abréviations* des options longues.

Cette fonctionnalité peut être désactivée en passant `False` à `allow_abbrev` :

```
>>> parser = argparse.ArgumentParser(prog='PROG', allow_abbrev=False)
>>> parser.add_argument('--foobar', action='store_true')
>>> parser.add_argument('--foonley', action='store_false')
>>> parser.parse_args(['--foon'])
usage: PROG [-h] [--foobar] [--foonley]
PROG: error: unrecognized arguments: --foon
```

Ajouté dans la version 3.5.

Le paramètre *conflict_handler*

Les objets `ArgumentParser` ne peuvent pas avoir plus d'une option avec la même chaîne d'option. Par défaut, les objets `ArgumentParser` lèvent une exception si on essaie de créer un argument avec une chaîne d'option qui est déjà utilisée :

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-f', '--foo', help='old foo help')
>>> parser.add_argument('--foo', help='new foo help')
Traceback (most recent call last):
 ..
ArgumentError: argument --foo: conflicting option string(s): --foo
```

Parfois, par exemple si on utilise des analyseurs *parents*, il est souhaitable de surcharger les anciens arguments qui partagent la même chaîne d'option. Pour obtenir ce comportement, vous devez passer `'resolve'` à l'argument `conflict_handler=` du constructeur d'`ArgumentParser` :

```
>>> parser = argparse.ArgumentParser(prog='PROG', conflict_handler='resolve')
>>> parser.add_argument('-f', '--foo', help='old foo help')
>>> parser.add_argument('--foo', help='new foo help')
>>> parser.print_help()
usage: PROG [-h] [-f FOO] [--foo FOO]

options:
 -h, --help  show this help message and exit
 -f FOO      old foo help
 --foo FOO   new foo help
```

Prenez note que les objets `ArgumentParser` n'enlèvent une action que si toutes ses chaînes d'options sont surchargées. Ainsi dans l'exemple ci-dessus, l'action `-f/--foo` du parent est conservée comme l'action `-f` puisque `--foo` est la seule chaîne d'options qui a été surchargée.

Le paramètre *add_help*

By default, `ArgumentParser` objects add an option which simply displays the parser's help message. If `-h` or `--help` is supplied at the command line, the `ArgumentParser` help will be printed.

Il est parfois utile de désactiver l'ajout de cette option d'aide. Pour ce faire, vous devez passer `False` à l'argument `add_help=` du constructeur d'`ArgumentParser` :

```
>>> parser = argparse.ArgumentParser(prog='PROG', add_help=False)
>>> parser.add_argument('--foo', help='foo help')
>>> parser.print_help()
usage: PROG [--foo FOO]

options:
 --foo FOO  foo help
```

En général, l'option d'aide est `-h/--help`. L'exception à cette règle est quand une valeur est passée à `prefix_chars=` et qu'elle n'inclue pas `-`, auquel cas, `-h` et `--help` ne sont pas des options valides. Dans ce cas, le

premier caractère de `prefix_chars` est utilisé comme préfixe des options d'aide :

```
>>> parser = argparse.ArgumentParser(prog='PROG', prefix_chars='+/')
>>> parser.print_help()
usage: PROG [+h]

options:
  +h, ++help  show this help message and exit
```

Le paramètre *exit_on_error*

Normally, when you pass an invalid argument list to the `parse_args()` method of an `ArgumentParser`, it will print a *message* to `sys.stderr` and exit with a status code of 2.

Si vous souhaitez intercepter les erreurs manuellement, la fonctionnalité peut être activée en assignant `False` à `exit_on_error` :

```
>>> parser = argparse.ArgumentParser(exit_on_error=False)
>>> parser.add_argument('--integers', type=int)
_StoreAction(option_strings=['--integers'], dest='integers', nargs=None,␣
↪const=None, default=None, type=<class 'int'>, choices=None, help=None,␣
↪metavar=None)
>>> try:
...     parser.parse_args('--integers a'.split())
... except argparse.ArgumentError:
...     print('Catching an argumentError')
...
Catching an argumentError
```

Ajouté dans la version 3.9.

suggest_on_error

By default, when a user passes an invalid argument choice or subparser name, `ArgumentParser` will exit with error info and list the permissible argument choices (if specified) or subparser names as part of the error message.

If the user would like to enable suggestions for mistyped argument choices and subparser names, the feature can be enabled by setting `suggest_on_error` to `True`. Note that this only applies for arguments when the choices specified are strings :

```
>>> parser = argparse.ArgumentParser(description='Process some integers.', suggest_
↪on_error=True)
>>> parser.add_argument('--action', choices=['sum', 'max'])
>>> parser.add_argument('integers', metavar='N', type=int, nargs='+',
...                     help='an integer for the accumulator')
>>> parser.parse_args(['--action', 'sumn', 1, 2, 3])
tester.py: error: argument --action: invalid choice: 'sumn', maybe you meant 'sum'?
↪ (choose from 'sum', 'max')
```

If you're writing code that needs to be compatible with older Python versions and want to opportunistically use `suggest_on_error` when it's available, you can set it as an attribute after initializing the parser instead of using the keyword argument :

```
>>> parser = argparse.ArgumentParser(description='Process some integers.')
>>> parser.suggest_on_error = True
```

Ajouté dans la version 3.14.

17.1.2 La méthode *add_argument()*

`ArgumentParser.add_argument`(*name or flags...*, *[, *action*][, *nargs*][, *const*][, *default*][, *type*][, *choices*][, *required*][, *help*][, *metavar*][, *dest*][, *deprecated*])

Définie comment une option de ligne de commande doit être analysée. Chacun des paramètres est décrit plus en détails ci-bas, mais en résumé ils sont :

- *name or flags* - Either a name or a list of option strings, e.g. `'foo'` or `'-f', '--foo'`.
- *action* – Type élémentaire de l'action à entreprendre quand cet argument est reconnu sur la ligne de commande ;
- *nargs* – Nombre d'arguments de la ligne de commande à capturer ;
- *const* – Valeur constante requise par certains choix d'*action* et de *nargs* ;
- *default* – Valeur produite si l'argument est absent de la ligne de commande et absent de l'objet `namespace` ;
- *type* – Type vers lequel l'argument sur la ligne de commande doit être converti ;
- *choices* – La séquence de valeurs autorisées pour cet argument ;
- *required* – `True` si l'option sur la ligne de commande est obligatoire (ne s'applique qu'aux arguments optionnels) ;
- *help* – Brève description de ce que fait l'argument ;
- *metavar* – Nom de l'argument dans les messages d'utilisations ;
- *dest* – Nom de l'attribut qui est ajouté à l'objet retourné par `parse_args()`.
- *deprecated* - Whether or not use of the argument is deprecated.

Les sections suivantes décrivent comment chacune de ces options sont utilisées.

Le paramètre *name_or_flags*

La méthode `add_argument()` doit savoir s'il s'agit d'un argument optionnel (tel que `-f` ou `--foo`) ou d'un argument positionnel (tel qu'une liste de noms de fichiers) qui est attendu. Le premier argument passé à `add_argument()` doit donc être soit une série de drapeaux, soit le nom de l'argument.

Par exemple, un argument optionnel est créé comme suit :

```
>>> parser.add_argument('-f', '--foo')
```

alors qu'un argument positionnel est créé comme suit :

```
>>> parser.add_argument('bar')
```

Lors le l'appel de `parse_args()`, les arguments qui commencent par le préfixe `-` sont présumés optionnels et tous les autres sont présumés positionnels :

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-f', '--foo')
>>> parser.add_argument('bar')
>>> parser.parse_args(['BAR'])
Namespace(bar='BAR', foo=None)
>>> parser.parse_args(['BAR', '--foo', 'FOO'])
Namespace(bar='BAR', foo='FOO')
>>> parser.parse_args(['--foo', 'FOO'])
usage: PROG [-h] [-f FOO] bar
PROG: error: the following arguments are required: bar
```

By default, `argparse` automatically handles the internal naming and display names of arguments, simplifying the process without requiring additional configuration. As such, you do not need to specify the *dest* and *metavar* parameters. The *dest* parameter defaults to the argument name with underscores `_` replacing hyphens `-` . The *metavar* parameter defaults to the upper-cased name. For example :

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo-bar')
>>> parser.parse_args(['--foo-bar', 'FOO-BAR']
```

(suite sur la page suivante)

(suite de la page précédente)

```
Namespace(foo_bar='FOO-BAR')
>>> parser.print_help()
usage:  [-h] [--foo-bar FOO-BAR]

optional arguments:
 -h, --help  show this help message and exit
 --foo-bar FOO-BAR
```

Le paramètre *action*

Les objets `ArgumentParser` associent les arguments de la ligne de commande avec des actions. Ces actions peuvent soumettre les arguments de la ligne de commande auxquels elles sont associées à un traitement arbitraire, mais la majorité des actions se contentent d'ajouter un attribut à l'objet renvoyé par `parse_args()`. L'argument nommé `action` indique comment l'argument de la ligne de commande est traité. Les actions natives sont :

— `'store'` - This just stores the argument's value. This is the default action.

— `'store_const'` – Stocke la valeur passée à l'argument nommé *const*. La valeur par défaut de *const* est `None`. L'action `'store_const'` est généralement utilisée avec des arguments optionnels représentant un drapeau ou une condition similaire. Par exemple :

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action='store_const', const=42)
>>> parser.parse_args(['--foo'])
Namespace(foo=42)
```

— `'store_true'` and `'store_false'` - These are special cases of `'store_const'` used for storing the values `True` and `False` respectively. In addition, they create default values of `False` and `True` respectively :

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action='store_true')
>>> parser.add_argument('--bar', action='store_false')
>>> parser.add_argument('--baz', action='store_false')
>>> parser.parse_args('--foo --bar'.split())
Namespace(foo=True, bar=False, baz=True)
```

— `'append'` – Stocke une liste et ajoute la valeur de son argument à cette liste. Il est donc généralement utile d'accepter la répétition de cet argument. Si une valeur par défaut est précisée, alors cette valeur est également présente dans la liste et précède les valeurs passées sur la ligne de commande :

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action='append')
>>> parser.parse_args('--foo 1 --foo 2'.split())
Namespace(foo=['1', '2'])
```

— `'append_const'` – Stocke une liste et ajoute la valeur de son argument *const* à cette liste. Notez que la valeur par défaut de l'argument *const* est `None`. L'action `'append_const'` est pratique quand plusieurs arguments ont besoin de stocker des constantes dans une seule liste. Par exemple :

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--str', dest='types', action='append_const',␣
↪const=str)
>>> parser.add_argument('--int', dest='types', action='append_const',␣
↪const=int)
>>> parser.parse_args('--str --int'.split())
Namespace(types=[<class 'str'>, <class 'int'>])
```

— `'extend'` - This stores a list and appends each item from the multi-value argument list to it. The `'extend'` action is typically used with the *nargs* keyword argument value `'+'` or `'*'`. Note that when *nargs* is `None` (the default) or `'?'`, each character of the argument string will be appended to the list. Example usage :

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument("--foo", action="extend", nargs="+", type=str)
>>> parser.parse_args(["--foo", "f1", "--foo", "f2", "f3", "f4"])
Namespace(foo=['f1', 'f2', 'f3', 'f4'])
```

Ajouté dans la version 3.8.

— `'count'` – Compte le nombre d'occurrences de l'argument nommé. Ceci est pratique, par exemple, pour augmenter le niveau de verbosité :

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--verbose', '-v', action='count', default=0)
>>> parser.parse_args(['-vvv'])
Namespace(verbose=3)
```

Prenez note que la valeur de *default* est `None` à moins qu'elle soit explicitement définie à `0`.

— `'help'` – Affiche le message d'aide complet pour toutes les options de l'analyseur puis termine l'exécution. Une action `help` est automatiquement ajoutée à l'analyseur par défaut. Consultez *`ArgumentParser`* pour les détails de la création du contenu de l'aide.

— `'version'` – Affiche la version du programme puis termine l'exécution. Cette action requiert l'argument nommé `version=` dans l'appel à *`add_argument()`* :

```
>>> import argparse
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--version', action='version', version='%(prog)s 2.0')
>>> parser.parse_args(['--version'])
PROG 2.0
```

Only actions that consume command-line arguments (e.g. `'store'`, `'append'` or `'extend'`) can be used with positional arguments.

class argparse.**BooleanOptionalAction**

You may also specify an arbitrary action by passing an *`Action`* subclass or other object that implements the same interface. The `BooleanOptionalAction` is available in `argparse` and adds support for boolean actions such as `--foo` and `--no-foo` :

```
>>> import argparse
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action=argparse.BooleanOptionalAction)
>>> parser.parse_args(['--no-foo'])
Namespace(foo=False)
```

Ajouté dans la version 3.9.

The recommended way to create a custom action is to extend *`Action`*, overriding the `__call__()` method and optionally the `__init__()` and `format_usage()` methods. You can also register custom actions using the *`register()`* method and reference them by their registered name.

Un exemple d'action personnalisée :

```
>>> class FooAction(argparse.Action):
...     def __init__(self, option_strings, dest, nargs=None, **kwargs):
...         if nargs is not None:
...             raise ValueError("nargs not allowed")
...         super().__init__(option_strings, dest, **kwargs)
...     def __call__(self, parser, namespace, values, option_string=None):
...         print('%r %r %r' % (namespace, values, option_string))
...         setattr(namespace, self.dest, values)
...
>>> parser = argparse.ArgumentParser()
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> parser.add_argument('--foo', action=FooAction)
>>> parser.add_argument('bar', action=FooAction)
>>> args = parser.parse_args('1 --foo 2'.split())
Namespace(bar=None, foo=None) '1' None
Namespace(bar='1', foo=None) '2' '--foo'
>>> args
Namespace(bar='1', foo='2')
```

Pour plus d'information, voir `Action`.

Le paramètre *nargs*

`ArgumentParser` objects usually associate a single command-line argument with a single action to be taken. The `nargs` keyword argument associates a different number of command-line arguments with a single action. See also *Specifying ambiguous arguments*. The supported values are :

— `N` (un entier). `N` arguments de la ligne de commande sont capturés ensemble et stockés dans une liste. Par exemple :

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', nargs=2)
>>> parser.add_argument('bar', nargs=1)
>>> parser.parse_args('c --foo a b'.split())
Namespace(bar=['c'], foo=['a', 'b'])
```

Prenez note que `nargs=1` produit une liste avec un seul élément. Ceci est différent du comportement par défaut qui produit l'élément directement (comme un scalaire).

— `'?'`. Un argument de la ligne de commande est capturé et produit directement. Si aucun argument n'est présent sur la ligne de commande, la valeur de *default* est produite. Prenez note que pour les arguments optionnels, il est aussi possible que la chaîne d'option soit présente mais qu'elle ne soit pas suivie d'un argument. Dans ce cas, la valeur de *const* est produite. Voici quelques exemples pour illustrer ceci :

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', nargs='?', const='c', default='d')
>>> parser.add_argument('bar', nargs='?', default='d')
>>> parser.parse_args(['XX', '--foo', 'YY'])
Namespace(bar='XX', foo='YY')
>>> parser.parse_args(['XX', '--foo'])
Namespace(bar='XX', foo='c')
>>> parser.parse_args([])
Namespace(bar='d', foo='d')
```

`nargs='?'` est fréquemment utilisé pour accepter des fichiers d'entrée et de sortie optionnels :

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('infile', nargs='?')
>>> parser.add_argument('outfile', nargs='?')
>>> parser.parse_args(['input.txt', 'output.txt'])
Namespace(infile='input.txt', outfile='output.txt')
>>> parser.parse_args(['input.txt'])
Namespace(infile='input.txt', outfile=None)
>>> parser.parse_args([])
Namespace(infile=None, outfile=None)
```

— `'*'`. Tous les arguments présents sur la ligne de commande sont capturés dans une liste. Prenez note qu'il n'est pas logique d'avoir plus d'un argument positionnel avec `nargs='*'`, mais il est par contre possible d'avoir plusieurs arguments optionnels qui spécifient `nargs='*'`. Par exemple :

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', nargs='*')
>>> parser.add_argument('--bar', nargs='*')
>>> parser.add_argument('baz', nargs='*')
>>> parser.parse_args('a b --foo x y --bar 1 2'.split())
Namespace(bar=['1', '2'], baz=['a', 'b'], foo=['x', 'y'])
```

— `'+'`. Comme pour `'*'`, tous les arguments présents sur la ligne de commande sont capturés dans une liste. De plus, un message d'erreur est produit s'il n'y a pas au moins un argument présent sur la ligne de commande. Par exemple :

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('foo', nargs='+')
>>> parser.parse_args(['a', 'b'])
Namespace(foo=['a', 'b'])
>>> parser.parse_args([])
usage: PROG [-h] foo [foo ...]
PROG: error: the following arguments are required: foo
```

If the `nargs` keyword argument is not provided, the number of arguments consumed is determined by the *action*. Generally this means a single command-line argument will be consumed and a single item (not a list) will be produced. Actions that do not consume command-line arguments (e.g. `'store_const'`) set `nargs=0`.

Le paramètre *const*

L'argument `const` d'*`add_argument()`* est utilisé pour stocker une constante qui n'est pas lue depuis la ligne de commande mais qui est requise par certaines actions d'*`ArgumentParser`*. Les deux utilisations les plus communes sont :

— quand *`add_argument()`* est appelée avec `action='store_const'` ou `action='append_const'`, ces actions ajoutent la valeur de `const` à l'un des attributs de l'objet renvoyé par *`parse_args()`*. Consultez la description d'*action* pour voir quelques exemples. La valeur de `const` est `None` par défaut pour *`add_argument()`* ;

— appeler la méthode *`add_argument()`* avec des chaînes d'options (telles `-f`, `--foo` ou avec `nargs='?'`) crée un argument optionnel auquel peut être attribué une valeur. Si celle-ci est omise, l'analyseur prend `None` comme la valeur de `const`. La description de *nargs* offre quelques exemples.

Modifié dans la version 3.11 : `const=None` par défaut, y compris avec `action='append_const'` ou `action='store_const'`.

Le paramètre *default*

Tous les arguments optionnels et certains arguments positionnels peuvent être omis à la ligne de commande. L'argument nommé `default` de la méthode *`add_argument()`* (qui vaut `None` par défaut), indique quelle valeur est utilisée si l'argument est absent de la ligne de commande. Pour les arguments optionnels, la valeur de `default` est utilisée si la chaîne d'option n'est pas présente sur la ligne de commande :

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default=42)
>>> parser.parse_args(['--foo', '2'])
Namespace(foo='2')
>>> parser.parse_args([])
Namespace(foo=42)
```

If the target namespace already has an attribute set, the action *default* will not overwrite it :

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default=42)
>>> parser.parse_args([], namespace=argparse.Namespace(foo=101))
Namespace(foo=101)
```

Si la valeur de `default` est une chaîne, l'analyseur analyse cette valeur comme si c'était un argument de la ligne de commande. En particulier, l'analyseur applique la conversion définie par l'argument *type* (si elle est fournie) avant d'affecter l'attribut à l'objet `Namespace` renvoyé. Autrement, l'analyseur utilise la valeur telle qu'elle :

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--length', default='10', type=int)
>>> parser.add_argument('--width', default=10.5, type=int)
>>> parser.parse_args()
Namespace(length=10, width=10.5)
```

Pour les arguments positionnels pour lesquels *nargs* est `?` ou `*`, la valeur de `default` est utilisée quand l'argument est absent de la ligne de commande :

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('foo', nargs='?', default=42)
>>> parser.parse_args(['a'])
Namespace(foo='a')
>>> parser.parse_args([])
Namespace(foo=42)
```

For *required* arguments, the `default` value is ignored. For example, this applies to positional arguments with *nargs* values other than `?` or `*`, or optional arguments marked as `required=True`.

Si vous passez `default=argparse.SUPPRESS`, aucun attribut n'est ajouté à l'objet `Namespace` quand l'argument est absent de la ligne de commande :

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default=argparse.SUPPRESS)
>>> parser.parse_args([])
Namespace()
>>> parser.parse_args(['--foo', '1'])
Namespace(foo='1')
```

Le paramètre *type*

Par défaut, l'analyseur capture les arguments de la ligne de commande comme des chaînes. Très souvent par contre, on désire interpréter les chaînes de la ligne de commande comme un autre type, tel que `float` ou `int`. L'argument nommé `type` d'`add_argument()` nous permet de faire les vérifications et les conversions de type nécessaires.

Si l'argument nommé *type* est utilisé en conjonction avec l'argument nommé *default*, le convertisseur de type n'est appliqué que si la valeur par défaut est une chaîne.

The argument to `type` can be a callable that accepts a single string or the name of a registered type (see `register()`) If the function raises `ArgumentTypeError`, `TypeError`, or `ValueError`, the exception is caught and a nicely formatted error message is displayed. Other exception types are not handled.

Les types et les fonctions natives peuvent être utilisés comme convertisseurs de types :

```
import argparse
import pathlib

parser = argparse.ArgumentParser()
parser.add_argument('count', type=int)
parser.add_argument('distance', type=float)
parser.add_argument('street', type=ascii)
parser.add_argument('code_point', type=ord)
parser.add_argument('datapath', type=pathlib.Path)
```

Des fonctions définies par l'utilisateur peuvent aussi être utilisées :

```
>>> def hyphenated(string):
...     return '-'.join([word[:4] for word in string.casefold().split()])
...
>>> parser = argparse.ArgumentParser()
>>> _ = parser.add_argument('short_title', type=hyphenated)
>>> parser.parse_args(['"The Tale of Two Cities"'])
Namespace(short_title='"the-tale-of-two-citi')
```

La fonction `bool()` est déconseillée comme convertisseur de type. Son comportement se limite à convertir les chaînes vides à `False` et les chaînes non-vides à `True`. Ce n'est généralement pas le résultat désiré.

En général, l'argument nommé `type` est un raccourci qui ne devrait être utilisé que pour les conversions simples qui ne peuvent lever qu'une des trois exceptions gérées. Les conversions qui demandent un traitement d'erreurs plus intéressant ou une gestion de ressources devraient être effectuées plus tard dans l'exécution suivant l'analyse des arguments.

Par exemple, les conversions de documents *JSON* ou *YAML* ont des cas d'erreurs complexes qui requièrent un meilleur signalement que ce qui est fourni par l'argument nommé `type`. Une exception `JSONDecodeError` ne serait pas adéquatement formatée et une exception `FileNotFoundError` ne serait pas du tout traitée.

Even `FileType` has its limitations for use with the `type` keyword. If one argument uses `FileType` and then a subsequent argument fails, an error is reported but the file is not automatically closed. In this case, it would be better to wait until after the parser has run and then use the `with`-statement to manage the files.

Pour les vérificateurs de types qui ne font que tester l'appartenance à un ensemble de valeurs, pensez plutôt à utiliser l'argument nommé *choices*.

Le paramètre *choices*

Certains arguments de la ligne de commande doivent être choisis parmi un ensemble prédéfini de valeurs. Celles-ci doivent être déclarées dans la séquence passée à l'argument *choices* de la méthode `add_argument()`. Un message d'erreur est alors affiché si l'utilisateur passe un argument qui n'est pas parmi les valeurs acceptables :

```
>>> parser = argparse.ArgumentParser(prog='game.py')
>>> parser.add_argument('move', choices=['rock', 'paper', 'scissors'])
>>> parser.parse_args(['rock'])
Namespace(move='rock')
>>> parser.parse_args(['fire'])
usage: game.py [-h] {rock,paper,scissors}
game.py: error: argument move: invalid choice: 'fire' (choose from 'rock',
'paper', 'scissors')
```

Note that inclusion in the *choices* sequence is checked after any *type* conversions have been performed, so the type of the objects in the *choices* sequence should match the *type* specified.

N'importe quel séquence peut être utilisée comme valeur de *choices*, dont les objets de type `list`, `set` ou des conteneurs personnalisés.

L'utilisation d'`enum.Enum` est déconseillée, car il est difficile de contrôler son apparence dans les messages d'usage, d'aide et d'erreur.

Les choix formatés écrasent la valeur par défaut de *metavar* qui est normalement dérivée de *dest*. C'est en général le comportement recherché car l'utilisateur ne voit jamais le paramètre *dest*. Si cet affichage n'est pas souhaité (comme lorsque les choix sont trop nombreux) spécifiez simplement *metavar* de façon explicite.

Le paramètre *required*

In general, the `argparse` module assumes that flags like `-f` and `--bar` indicate *optional* arguments, which can always be omitted at the command line. To make an option *required*, `True` can be specified for the `required=` keyword argument to `add_argument()` :

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', required=True)
>>> parser.parse_args(['--foo', 'BAR'])
Namespace(foo='BAR')
>>> parser.parse_args([])
usage: [-h] --foo FOO
: error: the following arguments are required: --foo
```

Tel qu'illustré dans l'exemple, quand l'option est marquée comme `required`, *`parse_args()`* mentionne une erreur si l'option est absente de la ligne de commande.

> **Note**
>
> En général, les options obligatoires manifestent un style boiteux, car les utilisateurs s'attendent à ce que les *options* soient *optionnelles*. Elles devraient donc être évitées si possible.

Le paramètre *help*

The `help` value is a string containing a brief description of the argument. When a user requests help (usually by using `-h` or `--help` at the command line), these `help` descriptions will be displayed with each argument.

La chaîne `help` peut contenir des définitions de formatage permettant d'éviter la répétition de contenu tel que le nom du programme et la valeur par défaut de l'argument (voir *default*). Les définitions disponibles comprennent entre autres le nom du programme, `%(prog)s`, et la plupart des arguments nommés d'*`add_argument()`*, tels que `%(default)s`, `%(type)s`, etc :

```
>>> parser = argparse.ArgumentParser(prog='frobble')
>>> parser.add_argument('bar', nargs='?', type=int, default=42,
...                     help='the bar to %(prog)s (default: %(default)s)')
>>> parser.print_help()
usage: frobble [-h] [bar]

positional arguments:
 bar     the bar to frobble (default: 42)

options:
 -h, --help  show this help message and exit
```

Comme la chaîne d'aide utilise le caractère `%` pour le formatage, si vous désirez afficher un `%` littéral dans la chaîne d'aide, vous devez en faire l'échappement avec `%%`.

`argparse` supports silencing the help entry for certain options, by setting the `help` value to `argparse.SUPPRESS` :

```
>>> parser = argparse.ArgumentParser(prog='frobble')
>>> parser.add_argument('--foo', help=argparse.SUPPRESS)
>>> parser.print_help()
usage: frobble [-h]

options:
  -h, --help  show this help message and exit
```

Le paramètre *metavar*

When *`ArgumentParser`* generates help messages, it needs some way to refer to each expected argument. By default, `ArgumentParser` objects use the *dest* value as the "name" of each object. By default, for positional argument actions, the *dest* value is used directly, and for optional argument actions, the *dest* value is uppercased. So, a single positional argument with `dest='bar'` will be referred to as `bar`. A single optional argument `--foo` that should be followed by a single command-line argument will be referred to as `FOO`. An example :

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> parser.add_argument('bar')
>>> parser.parse_args('X --foo Y'.split())
Namespace(bar='X', foo='Y')
>>> parser.print_help()
usage:  [-h] [--foo FOO] bar

positional arguments:
 bar

options:
 -h, --help  show this help message and exit
 --foo FOO
```

Un nom alternatif peut être fourni à `metavar` :

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', metavar='YYY')
>>> parser.add_argument('bar', metavar='XXX')
>>> parser.parse_args('X --foo Y'.split())
Namespace(bar='X', foo='Y')
>>> parser.print_help()
usage:  [-h] [--foo YYY] XXX

positional arguments:
 XXX

options:
 -h, --help  show this help message and exit
 --foo YYY
```

Prenez note que `metavar` ne change que le nom *affiché*. Le nom de l'attribut ajouté à l'objet renvoyé par `parse_args()` est toujours déterminé par la valeur de *dest*.

Certaines valeurs de `nargs` peuvent provoquer l'affichage de `metavar` plus d'une fois. Passer un *n-uplet* à `metavar` indique les différents noms à afficher pour chacun des arguments :

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x', nargs=2)
>>> parser.add_argument('--foo', nargs=2, metavar=('bar', 'baz'))
>>> parser.print_help()
usage: PROG [-h] [-x X X] [--foo bar baz]

options:
 -h, --help     show this help message and exit
 -x X X
 --foo bar baz
```

Le paramètre *dest*

La plupart des actions d'`ArgumentParser` ajoutent une valeur dans un attribut de l'objet renvoyé par `parse_args()`. Le nom de l'attribut est déterminé par l'argument nommé `dest` d'`add_argument()`. Pour les arguments positionnels, `dest` est généralement le premier argument d'`add_argument()` :

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('bar')
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> parser.parse_args(['XXX'])
Namespace(bar='XXX')
```

Pour les actions d'arguments optionnels, la valeur de `dest` est généralement inférée des chaînes d'options. *ArgumentParser* génère la valeur de `dest` en retirant le préfixe `--` de la première chaîne d'options longues. Si aucune n'est fournie, `dest` est alors dérivée de la première chaîne d'options courtes sans le préfixe `-`. Tous les `-` suivants sont convertis en _ pour s'assurer que la chaîne est un nom d'attribut valide. Les exemples suivants illustrent ce comportement :

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('-f', '--foo-bar', '--foo')
>>> parser.add_argument('-x', '-y')
>>> parser.parse_args('-f 1 -x 2'.split())
Namespace(foo_bar='1', x='2')
>>> parser.parse_args('--foo 1 -y 2'.split())
Namespace(foo_bar='1', x='2')
```

`dest` vous permet de fournir un nom d'attribut personnalisé :

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', dest='bar')
>>> parser.parse_args('--foo XXX'.split())
Namespace(bar='XXX')
```

deprecated

During a project's lifetime, some arguments may need to be removed from the command line. Before removing them, you should inform your users that the arguments are deprecated and will be removed. The `deprecated` keyword argument of *add_argument()*, which defaults to `False`, specifies if the argument is deprecated and will be removed in the future. For arguments, if `deprecated` is `True`, then a warning will be printed to *sys.stderr* when the argument is used :

```
>>> import argparse
>>> parser = argparse.ArgumentParser(prog='snake.py')
>>> parser.add_argument('--legs', default=0, type=int, deprecated=True)
>>> parser.parse_args([])
Namespace(legs=0)
>>> parser.parse_args(['--legs', '4'])
snake.py: warning: option '--legs' is deprecated
Namespace(legs=4)
```

Ajouté dans la version 3.13.

Classes Action

`Action` classes implement the Action API, a callable which returns a callable which processes arguments from the command-line. Any object which follows this API may be passed as the `action` parameter to *add_argument()*.

class argparse.**Action**(*option_strings*, *dest*, *nargs=None*, *const=None*, *default=None*, *type=None*, *choices=None*, *required=False*, *help=None*, *metavar=None*)

`Action` objects are used by an *ArgumentParser* to represent the information needed to parse a single argument from one or more strings from the command line. The `Action` class must accept the two positional arguments plus any keyword arguments passed to *ArgumentParser.add_argument()* except for the `action` itself.

Instances of `Action` (or return value of any callable to the `action` parameter) should have attributes `dest`, `option_strings`, `default`, `type`, `required`, `help`, etc. defined. The easiest way to ensure these attributes are defined is to call `Action.__init__()`.

__call__(*parser*, *namespace*, *values*, *option_string=None*)

`Action` instances should be callable, so subclasses must override the `__call__()` method, which should accept four parameters :

— *parser* - The `ArgumentParser` object which contains this action.

— *namespace* - The `Namespace` object that will be returned by `parse_args()`. Most actions add an attribute to this object using `setattr()`.

— *values* - The associated command-line arguments, with any type conversions applied. Type conversions are specified with the *type* keyword argument to `add_argument()`.

— *option_string* - The option string that was used to invoke this action. The `option_string` argument is optional, and will be absent if the action is associated with a positional argument.

The `__call__()` method may perform arbitrary actions, but will typically set attributes on the `namespace` based on `dest` and `values`.

format_usage()

`Action` subclasses can define a `format_usage()` method that takes no argument and return a string which will be used when printing the usage of the program. If such method is not provided, a sensible default will be used.

17.1.3 La méthode *parse_args()*

`ArgumentParser.`**parse_args**(*args=None*, *namespace=None*)

Convertie les chaînes d'arguments en objets et les assigne comme attributs de l'espace de nommage `namespace`. Renvoie un objet `namespace` rempli.

Previous calls to `add_argument()` determine exactly what objects are created and how they are assigned. See the documentation for `add_argument()` for details.

— *args* – Liste de chaînes à analyser. La valeur par défaut est récupérée dans `sys.argv`.

— *namespace* – Un objet pour recevoir les attributs. Par défaut, une nouvelle instance (vide) de `Namespace`.

Syntaxe de la valeur des options

La méthode `parse_args()` offre plusieurs façons d'indiquer la valeur d'une option si elle en prend une. Dans le cas le plus simple, l'option et sa valeur sont passées en tant que deux arguments distincts :

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x')
>>> parser.add_argument('--foo')
>>> parser.parse_args(['-x', 'X'])
Namespace(foo=None, x='X')
>>> parser.parse_args(['--foo', 'FOO'])
Namespace(foo='FOO', x=None)
```

Pour les options longues (les options qui ont un nom de plus d'un caractère), l'option et sa valeur peuvent être passées comme un seul argument de la ligne de commande en utilisant = comme séparateur :

```
>>> parser.parse_args(['--foo=FOO'])
Namespace(foo='FOO', x=None)
```

Pour les options courtes (un seul caractère), l'option et sa valeur peuvent être concaténées :

```
>>> parser.parse_args(['-xX'])
Namespace(foo=None, x='X')
```

Plusieurs options courtes peuvent être groupées ensemble après un seul préfixe - pour autant que seule la dernière, au maximum, ne nécessite une valeur :

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x', action='store_true')
>>> parser.add_argument('-y', action='store_true')
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> parser.add_argument('-z')
>>> parser.parse_args(['-xyzZ'])
Namespace(x=True, y=True, z='Z')
```

Arguments invalides

Quand elle fait l'analyse de la ligne de commande, la méthode *parse_args()* vérifie plusieurs erreurs possibles : entre autres, options ambigües, types invalides, options invalides, nombre incorrect d'arguments positionnels, etc. Quand une erreur est rencontrée, elle affiche l'erreur accompagnée du message d'aide puis termine l'exécution :

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo', type=int)
>>> parser.add_argument('bar', nargs='?')

>>> # invalid type
>>> parser.parse_args(['--foo', 'spam'])
usage: PROG [-h] [--foo FOO] [bar]
PROG: error: argument --foo: invalid int value: 'spam'

>>> # invalid option
>>> parser.parse_args(['--bar'])
usage: PROG [-h] [--foo FOO] [bar]
PROG: error: no such option: --bar

>>> # wrong number of arguments
>>> parser.parse_args(['spam', 'badger'])
usage: PROG [-h] [--foo FOO] [bar]
PROG: error: extra arguments found: badger
```

Arguments contenant –

La méthode *parse_args()* tente de signaler une erreur quand l'utilisateur s'est clairement trompé. Par contre, certaines situations sont intrinsèquement ambigües. Par exemple, l'argument de la ligne de commande `-1` peut aussi bien être une tentative de spécifier une option qu'une tentative de passer un argument positionnel. La méthode *parse_args()* est prudente : les arguments positionnels ne peuvent commencer par `-` que s'ils ont l'apparence d'un nombre négatif et que l'analyseur ne contient aucune option qui a l'apparence d'un nombre négatif :

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x')
>>> parser.add_argument('foo', nargs='?')

>>> # no negative number options, so -1 is a positional argument
>>> parser.parse_args(['-x', '-1'])
Namespace(foo=None, x='-1')

>>> # no negative number options, so -1 and -5 are positional arguments
>>> parser.parse_args(['-x', '-1', '-5'])
Namespace(foo='-5', x='-1')

>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-1', dest='one')
>>> parser.add_argument('foo', nargs='?')

>>> # negative number options present, so -1 is an option
>>> parser.parse_args(['-1', 'X'])
Namespace(foo=None, one='X')
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> # negative number options present, so -2 is an option
>>> parser.parse_args(['-2'])
usage: PROG [-h] [-1 ONE] [foo]
PROG: error: no such option: -2

>>> # negative number options present, so both -1s are options
>>> parser.parse_args(['-1', '-1'])
usage: PROG [-h] [-1 ONE] [foo]
PROG: error: argument -1: expected one argument
```

Si l'utilisateur a des arguments positionnels qui commencent par - et qui n'ont pas l'apparence d'un nombre négatif, il peut insérer le pseudo-argument `'--'` qui indique à `parse_args()` de traiter tout ce qui suit comme un argument positionnel :

```
>>> parser.parse_args(['--', '-f'])
Namespace(foo='-f', one=None)
```

See also *the argparse howto on ambiguous arguments* for more details.

Arguments abrégés (par comparaison de leurs préfixes)

Par défaut, la méthode `parse_args()` accepte que les options longues soient *abrégées* par un préfixe pour autant que l'abréviation soit non-ambigüe, c'est-à-dire qu'elle ne corresponde à aucune autre option :

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-bacon')
>>> parser.add_argument('-badger')
>>> parser.parse_args('-bac MMM'.split())
Namespace(bacon='MMM', badger=None)
>>> parser.parse_args('-bad WOOD'.split())
Namespace(bacon=None, badger='WOOD')
>>> parser.parse_args('-ba BA'.split())
usage: PROG [-h] [-bacon BACON] [-badger BADGER]
PROG: error: ambiguous option: -ba could match -badger, -bacon
```

Une erreur est générée pour les arguments qui peuvent produire plus d'une option. Ce comportement peut être désactivé en passant `False` à *Le paramètre allow_abbrev*.

Au-delà de `sys.argv`

Sometimes it may be useful to have an `ArgumentParser` parse arguments other than those of `sys.argv`. This can be accomplished by passing a list of strings to `parse_args()`. This is useful for testing at the interactive prompt :

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument(
...     'integers', metavar='int', type=int, choices=range(10),
...     nargs='+', help='an integer in the range 0..9')
>>> parser.add_argument(
...     '--sum', dest='accumulate', action='store_const', const=sum,
...     default=max, help='sum the integers (default: find the max)')
>>> parser.parse_args(['1', '2', '3', '4'])
Namespace(accumulate=<built-in function max>, integers=[1, 2, 3, 4])
>>> parser.parse_args(['1', '2', '3', '4', '--sum'])
Namespace(accumulate=<built-in function sum>, integers=[1, 2, 3, 4])
```

L'objet `Namespace`

class argparse.**Namespace**

Classe rudimentaire utilisée par défaut par *parse_args()* pour créer un objet qui stocke les attributs. Cet objet est renvoyé par `ArgumentParser.parse_args`.

Cette classe est délibérément rudimentaire : une sous-classe d'*object* avec une représentation textuelle intelligible. Si vous préférez une vue *dict-compatible*, vous devez utiliser *vars()* (un idiome Python classique) :

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> args = parser.parse_args(['--foo', 'BAR'])
>>> vars(args)
{'foo': 'BAR'}
```

Il est parfois utile de demander à *ArgumentParser* de faire l'affectation des attributs sur un objet existant plutôt que de faire la création d'un nouvel objet *Namespace*. Ceci peut être réalisé avec l'argument nommé `namespace=` :

```
>>> class C:
...     pass
...
>>> c = C()
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> parser.parse_args(args=['--foo', 'BAR'], namespace=c)
>>> c.foo
'BAR'
```

17.1.4 Autres outils

Sous commandes

ArgumentParser.**add_subparsers**(*[, *title*][, *description*][, *prog*][, *parser_class*][, *action*][, *dest*][, *required*][, *help*][, *metavar*])

Many programs split up their functionality into a number of subcommands, for example, the `svn` program can invoke subcommands like `svn checkout`, `svn update`, and `svn commit`. Splitting up functionality this way can be a particularly good idea when a program performs several different functions which require different kinds of command-line arguments. *ArgumentParser* supports the creation of such subcommands with the `add_subparsers()` method. The `add_subparsers()` method is normally called with no arguments and returns a special action object. This object has a single method, `add_parser()`, which takes a command name and any `ArgumentParser` constructor arguments, and returns an `ArgumentParser` object that can be modified as usual.

Description des paramètres :

- *title* - title for the sub-parser group in help output ; by default "subcommands" if description is provided, otherwise uses title for positional arguments
- *description* - description for the sub-parser group in help output, by default `None`
- *prog* - usage information that will be displayed with sub-command help, by default the name of the program and any positional arguments before the subparser argument
- *parser_class* - class which will be used to create sub-parser instances, by default the class of the current parser (e.g. *ArgumentParser*)
- *action* – action à entreprendre quand cet argument est reconnu sur la ligne de commande ;
- *dest* – nom de l'attribut sous lequel la sous-commande est stockée ; par défaut : `None` et aucune valeur n'est stockée ;
- *required* – `True` si la sous-commande est obligatoire ; par défaut : `False` (ajouté dans 3.7) ;
- *help* – message d'aide pour le groupe du sous-analyseur dans la sortie d'aide ; par défaut : `None` ;
- *metavar* - string presenting available subcommands in help ; by default it is `None` and presents subcommands in form {cmd1, cmd2, ..}

Quelques exemples d'utilisation :

```
>>> # create the top-level parser
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo', action='store_true', help='foo help')
>>> subparsers = parser.add_subparsers(help='subcommand help')
>>>
>>> # create the parser for the "a" command
>>> parser_a = subparsers.add_parser('a', help='a help')
>>> parser_a.add_argument('bar', type=int, help='bar help')
>>>
>>> # create the parser for the "b" command
>>> parser_b = subparsers.add_parser('b', help='b help')
>>> parser_b.add_argument('--baz', choices=('X', 'Y', 'Z'), help='baz help')
>>>
>>> # parse some argument lists
>>> parser.parse_args(['a', '12'])
Namespace(bar=12, foo=False)
>>> parser.parse_args(['--foo', 'b', '--baz', 'Z'])
Namespace(baz='Z', foo=True)
```

Prenez note que l'objet renvoyé par `parse_args()` ne contient que les attributs reconnus par l'analyseur principal et le sous-analyseur sélectionné par la ligne de commande. Les autres sous-analyseurs n'ont pas d'influence sur l'objet renvoyé. Ainsi dans l'exemple précédent, quand la commande `a` est donnée, seuls les attributs `foo` et `bar` sont présents alors que la commande `b` présente les attributs `foo` et `baz`.

Similarly, when a help message is requested from a subparser, only the help for that particular parser will be printed. The help message will not include parent parser or sibling parser messages. (A help message for each subparser command, however, can be given by supplying the `help=` argument to `add_parser()` as above.)

```
>>> parser.parse_args(['--help'])
usage: PROG [-h] [--foo] {a,b} ...

positional arguments:
  {a,b}   subcommand help
    a     a help
    b     b help

options:
  -h, --help  show this help message and exit
  --foo   foo help

>>> parser.parse_args(['a', '--help'])
usage: PROG a [-h] bar

positional arguments:
  bar     bar help

options:
  -h, --help  show this help message and exit

>>> parser.parse_args(['b', '--help'])
usage: PROG b [-h] [--baz {X,Y,Z}]

options:
  -h, --help     show this help message and exit
  --baz {X,Y,Z}  baz help
```

La méthode `add_subparsers()` accepte les arguments nommés `title` et `description`. Quand au moins

l'un des deux est présent, les commandes du sous-analyseur sont affichées dans leur propre groupe dans la sortie d'aide. Par exemple :

```
>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers(title='subcommands',
...                                    description='valid subcommands',
...                                    help='additional help')
>>> subparsers.add_parser('foo')
>>> subparsers.add_parser('bar')
>>> parser.parse_args(['-h'])
usage:  [-h] {foo,bar} ...

options:
  -h, --help  show this help message and exit

subcommands:
  valid subcommands

  {foo,bar}   additional help
```

Furthermore, `add_parser()` supports an additional *aliases* argument, which allows multiple strings to refer to the same subparser. This example, like `svn`, aliases `co` as a shorthand for `checkout` :

```
>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers()
>>> checkout = subparsers.add_parser('checkout', aliases=['co'])
>>> checkout.add_argument('foo')
>>> parser.parse_args(['co', 'bar'])
Namespace(foo='bar')
```

`add_parser()` supports also an additional *deprecated* argument, which allows to deprecate the subparser.

```
>>> import argparse
>>> parser = argparse.ArgumentParser(prog='chicken.py')
>>> subparsers = parser.add_subparsers()
>>> run = subparsers.add_parser('run')
>>> fly = subparsers.add_parser('fly', deprecated=True)
>>> parser.parse_args(['fly'])
chicken.py: warning: command 'fly' is deprecated
Namespace()
```

Ajouté dans la version 3.13.

One particularly effective way of handling subcommands is to combine the use of the *add_subparsers()* method with calls to *set_defaults()* so that each subparser knows which Python function it should execute. For example :

```
>>> # subcommand functions
>>> def foo(args):
...     print(args.x * args.y)
...
>>> def bar(args):
...     print('((%s))' % args.z)
...
>>> # create the top-level parser
>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers(required=True)
>>>
>>> # create the parser for the "foo" command
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> parser_foo = subparsers.add_parser('foo')
>>> parser_foo.add_argument('-x', type=int, default=1)
>>> parser_foo.add_argument('y', type=float)
>>> parser_foo.set_defaults(func=foo)
>>>
>>> # create the parser for the "bar" command
>>> parser_bar = subparsers.add_parser('bar')
>>> parser_bar.add_argument('z')
>>> parser_bar.set_defaults(func=bar)
>>>
>>> # parse the args and call whatever function was selected
>>> args = parser.parse_args('foo 1 -x 2'.split())
>>> args.func(args)
2.0
>>>
>>> # parse the args and call whatever function was selected
>>> args = parser.parse_args('bar XYZYX'.split())
>>> args.func(args)
((XYZYX))
```

Ainsi, vous pouvez laisser à `parse_args()` la responsabilité de faire appel à la bonne fonction après avoir analysé les arguments. Associer fonctions et actions est en général la façon la plus facile de gérer des actions différentes pour chacun de vos sous-analyseurs. Par contre, si vous avez besoin de consulter le nom du sous-analyseur qui a été invoqué, vous pouvez utiliser l'argument nommé `dest` d'`add_subparsers()` :

```
>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers(dest='subparser_name')
>>> subparser1 = subparsers.add_parser('1')
>>> subparser1.add_argument('-x')
>>> subparser2 = subparsers.add_parser('2')
>>> subparser2.add_argument('y')
>>> parser.parse_args(['2', 'frobble'])
Namespace(subparser_name='2', y='frobble')
```

Modifié dans la version 3.7 : New *required* keyword-only parameter.

Modifié dans la version 3.14 : Subparser's *prog* is no longer affected by a custom usage message in the main parser.

Objets `FileType`

class argparse.**FileType**(*mode='r'*, *bufsize=-1*, *encoding=None*, *errors=None*)

Le type fabrique `FileType` crée des objets qui peuvent être passés à l'argument `type` d'`ArgumentParser.add_argument()`. Les arguments qui ont comme `type` un objet `FileType` ouvrent les arguments de la ligne de commande en tant que fichiers avec les options spécifiées : mode, taille du tampon, encodage et gestion des erreurs (voir la fonction `open()` pour plus de détails) :

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--raw', type=argparse.FileType('wb', 0))
>>> parser.add_argument('out', type=argparse.FileType('w', encoding='UTF-8'))
>>> parser.parse_args(['--raw', 'raw.dat', 'file.txt'])
Namespace(out=<_io.TextIOWrapper name='file.txt' mode='w' encoding='UTF-8'>,␣
↪raw=<_io.FileIO name='raw.dat' mode='wb'>)
```

Les objets `FileType` reconnaissent le pseudo-argument `'-'` et le convertissent automatiquement vers `sys.stdin` pour les objets `FileType` ouverts en lecture, et vers `sys.stdout` pour les objets `FileType` ouverts en écriture :

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('infile', type=argparse.FileType('r'))
>>> parser.parse_args(['-'])
Namespace(infile=<_io.TextIOWrapper name='<stdin>' encoding='UTF-8'>)
```

> **Note**
>
> If one argument uses *FileType* and then a subsequent argument fails, an error is reported but the file is not automatically closed. This can also clobber the output files. In this case, it would be better to wait until after the parser has run and then use the `with`-statement to manage the files.

Modifié dans la version 3.4 : Added the *encodings* and *errors* parameters.

Obsolète depuis la version 3.14.

Groupes d'arguments

`ArgumentParser.`**`add_argument_group`**(*title=None*, *description=None*, *[, *argument_default*][, *conflict_handler*])

By default, `ArgumentParser` groups command-line arguments into "positional arguments" and "options" when displaying help messages. When there is a better conceptual grouping of arguments than this default one, appropriate groups can be created using the `add_argument_group()` method :

```
>>> parser = argparse.ArgumentParser(prog='PROG', add_help=False)
>>> group = parser.add_argument_group('group')
>>> group.add_argument('--foo', help='foo help')
>>> group.add_argument('bar', help='bar help')
>>> parser.print_help()
usage: PROG [--foo FOO] bar

group:
  bar    bar help
  --foo FOO  foo help
```

The `add_argument_group()` method returns an argument group object which has an `add_argument()` method just like a regular `ArgumentParser`. When an argument is added to the group, the parser treats it just like a normal argument, but displays the argument in a separate group for help messages. The `add_argument_group()` method accepts *title* and *description* arguments which can be used to customize this display :

```
>>> parser = argparse.ArgumentParser(prog='PROG', add_help=False)
>>> group1 = parser.add_argument_group('group1', 'group1 description')
>>> group1.add_argument('foo', help='foo help')
>>> group2 = parser.add_argument_group('group2', 'group2 description')
>>> group2.add_argument('--bar', help='bar help')
>>> parser.print_help()
usage: PROG [--bar BAR] foo

group1:
  group1 description

  foo    foo help

group2:
  group2 description
```

(suite sur la page suivante)

(suite de la page précédente)

```
  --bar BAR  bar help
```

The optional, keyword-only parameters *argument_default* and *conflict_handler* allow for finer-grained control of the behavior of the argument group. These parameters have the same meaning as in the `ArgumentParser` constructor, but apply specifically to the argument group rather than the entire parser.

Prenez note que tout argument qui n'est pas dans l'un de vos groupes est affiché dans l'une des sections usuelles *positional arguments* et *optional arguments*.

Deprecated since version 3.11, removed in version 3.14 : Calling `add_argument_group()` on an argument group now raises an exception. This nesting was never supported, often failed to work correctly, and was unintentionally exposed through inheritance.

Obsolète depuis la version 3.14 : Passing *prefix_chars* to `add_argument_group()` is now deprecated.

Exclusion mutuelle

`ArgumentParser.`**`add_mutually_exclusive_group`**(*required=False*)

Create a mutually exclusive group. `argparse` will make sure that only one of the arguments in the mutually exclusive group was present on the command line :

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> group = parser.add_mutually_exclusive_group()
>>> group.add_argument('--foo', action='store_true')
>>> group.add_argument('--bar', action='store_false')
>>> parser.parse_args(['--foo'])
Namespace(bar=True, foo=True)
>>> parser.parse_args(['--bar'])
Namespace(bar=False, foo=False)
>>> parser.parse_args(['--foo', '--bar'])
usage: PROG [-h] [--foo | --bar]
PROG: error: argument --bar: not allowed with argument --foo
```

La méthode `add_mutually_exclusive_group()` accepte aussi l'argument `required` pour indiquer qu'au moins un des arguments mutuellement exclusifs est nécessaire :

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> group = parser.add_mutually_exclusive_group(required=True)
>>> group.add_argument('--foo', action='store_true')
>>> group.add_argument('--bar', action='store_false')
>>> parser.parse_args([])
usage: PROG [-h] (--foo | --bar)
PROG: error: one of the arguments --foo --bar is required
```

Note that currently mutually exclusive argument groups do not support the *title* and *description* arguments of `add_argument_group()`. However, a mutually exclusive group can be added to an argument group that has a title and description. For example :

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> group = parser.add_argument_group('Group title', 'Group description')
>>> exclusive_group = group.add_mutually_exclusive_group(required=True)
>>> exclusive_group.add_argument('--foo', help='foo help')
>>> exclusive_group.add_argument('--bar', help='bar help')
>>> parser.print_help()
usage: PROG [-h] (--foo FOO | --bar BAR)

options:
  -h, --help  show this help message and exit
```

(suite sur la page suivante)

(suite de la page précédente)

```
Group title:
  Group description

  --foo FOO   foo help
  --bar BAR   bar help
```

Deprecated since version 3.11, removed in version 3.14 : Calling `add_argument_group()` or `add_mutually_exclusive_group()` on a mutually exclusive group now raises an exception. This nesting was never supported, often failed to work correctly, and was unintentionally exposed through inheritance.

Valeurs par défaut de l'analyseur

`ArgumentParser.`**`set_defaults`**(***kwargs*)

Dans la majorité des cas, les attributs de l'objet renvoyé par `parse_args()` sont entièrement définis par l'inspection des arguments de la ligne de commande et par les actions des arguments. La méthode `set_defaults()` permet l'ajout d'attributs additionnels qui sont définis sans nécessiter l'inspection de la ligne de commande :

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('foo', type=int)
>>> parser.set_defaults(bar=42, baz='badger')
>>> parser.parse_args(['736'])
Namespace(bar=42, baz='badger', foo=736)
```

Prenez note que les valeurs par défaut au niveau de l'analyseur ont précédence sur les valeurs par défaut au niveau de l'argument :

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default='bar')
>>> parser.set_defaults(foo='spam')
>>> parser.parse_args([])
Namespace(foo='spam')
```

Les valeurs par défaut au niveau de l'analyseur sont particulièrement utiles quand on travaille avec plusieurs analyseurs. Voir la méthode `add_subparsers()` pour un exemple de cette utilisation.

`ArgumentParser.`**`get_default`**(*dest*)

Renvoie la valeur par défaut d'un attribut de l'objet `Namespace` tel qu'il a été défini soit par `add_argument()` ou par `set_defaults()` :

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default='badger')
>>> parser.get_default('foo')
'badger'
```

Afficher l'aide

Pour la majorité des applications, `parse_args()` se charge du formatage et de l'affichage des messages d'erreur et d'utilisation. Plusieurs méthodes de formatage sont toutefois disponibles :

`ArgumentParser.`**`print_usage`**(*file=None*)

Affiche une brève description sur la façon d'invoquer l'`ArgumentParser` depuis la ligne de commande. Si `file` est `None`, utilise `sys.stdout`.

`ArgumentParser.`**`print_help`**(*file=None*)

Affiche un message d'aide qui inclut l'utilisation du programme et l'information sur les arguments répertoriés dans l'`ArgumentParser`. Si `file` est `None`, utilise `sys.stdout`.

Des variantes de ces méthodes sont fournies pour renvoyer la chaîne plutôt que de l'afficher :

`ArgumentParser.format_usage()`

Renvoie une chaîne contenant une brève description sur la façon d'invoquer l'`ArgumentParser` depuis la ligne de commande.

`ArgumentParser.format_help()`

Renvoie une chaîne représentant un message d'aide qui inclut des informations sur l'utilisation du programme et sur les arguments définis dans l'`ArgumentParser`.

Analyse partielle

`ArgumentParser.parse_known_args(args=None, namespace=None)`

Parfois, un script n'analyse que quelques-uns des arguments de la ligne de commande avant de passer les arguments non-traités à un autre script ou programme. La méthode `parse_known_args()` est utile dans ces cas. Elle fonctionne similairement à `parse_args()`, mais elle ne lève pas d'erreur quand des arguments non-reconnus sont présents. Au lieu, elle renvoie une paire de valeurs : l'objet `Namespace` rempli et la liste des arguments non-traités.

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action='store_true')
>>> parser.add_argument('bar')
>>> parser.parse_known_args(['--foo', '--badger', 'BAR', 'spam'])
(Namespace(bar='BAR', foo=True), ['--badger', 'spam'])
```

Avertissement

Les règles d'*acceptation des abréviations* sont applicables à `parse_known_args()`. L'analyseur peut ainsi capturer une option même si elle n'est que le préfixe d'une option reconnue plutôt que de la laisser dans la liste des arguments non-traités.

Personnaliser le *parsing* de fichiers

`ArgumentParser.convert_arg_line_to_args(arg_line)`

Les arguments qui proviennent d'un fichier sont lus un par ligne. La méthode `convert_arg_line_to_args()` peut être surchargée pour accomplir un traitement plus élaboré. Voir aussi l'argument nommé `fromfile_prefix_chars` du constructeur d'`ArgumentParser`.

La méthode `convert_arg_line_to_args` accepte un seul argument, `arg_line`, qui est une chaîne lue dans le fichier d'arguments. Elle renvoie une liste d'arguments analysés dans cette chaîne. La méthode est appelée une fois pour chaque ligne lue du fichier d'arguments. L'ordre est préservé.

Une surcharge utile de cette méthode est de permettre à chaque mot délimité par des espaces d'être traité comme un argument. L'exemple suivant illustre comment réaliser ceci :

```
class MyArgumentParser(argparse.ArgumentParser):
    def convert_arg_line_to_args(self, arg_line):
        return arg_line.split()
```

Méthodes d'interruptions

`ArgumentParser.exit(status=0, message=None)`

This method terminates the program, exiting with the specified *status* and, if given, it prints a *message* to `sys.stderr` before that. The user can override this method to handle these steps differently :

```
class ErrorCatchingArgumentParser(argparse.ArgumentParser):
    def exit(self, status=0, message=None):
```

(suite sur la page suivante)

(suite de la page précédente)

```
        if status:
            raise Exception(f'Exiting because of an error: {message}')
        exit(status)
```

`ArgumentParser.error(message)`

This method prints a usage message, including the *message*, to `sys.stderr` and terminates the program with a status code of 2.

Analyse entremêlée

`ArgumentParser.parse_intermixed_args(args=None, namespace=None)`

`ArgumentParser.parse_known_intermixed_args(args=None, namespace=None)`

De nombreuses commandes Unix permettent à l'utilisateur d'entremêler les arguments optionnels et les arguments positionnels. Les méthodes `parse_intermixed_args()` et `parse_known_intermixed_args()` permettent ce style d'analyse.

These parsers do not support all the `argparse` features, and will raise exceptions if unsupported features are used. In particular, subparsers, and mutually exclusive groups that include both optionals and positionals are not supported.

L'exemple suivant illustre la différence entre `parse_known_args()` et `parse_intermixed_args()` : le premier renvoie `['2', '3']` comme arguments non-traités alors que le second capture tous les arguments positionnels dans `rest`

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> parser.add_argument('cmd')
>>> parser.add_argument('rest', nargs='*', type=int)
>>> parser.parse_known_args('doit 1 --foo bar 2 3'.split())
(Namespace(cmd='doit', foo='bar', rest=[1]), ['2', '3'])
>>> parser.parse_intermixed_args('doit 1 --foo bar 2 3'.split())
Namespace(cmd='doit', foo='bar', rest=[1, 2, 3])
```

`parse_known_intermixed_args()` renvoie une paire de valeurs : l'objet `Namespace` rempli et une liste de chaînes d'arguments non-traités. `parse_intermixed_args()` lève une erreur s'il reste des chaînes d'arguments non-traités.

Ajouté dans la version 3.7.

Registering custom types or actions

`ArgumentParser.register(registry_name, value, object)`

Sometimes it's desirable to use a custom string in error messages to provide more user-friendly output. In these cases, `register()` can be used to register custom actions or types with a parser and allow you to reference the type by their registered name instead of their callable name.

The `register()` method accepts three arguments - a *registry_name*, specifying the internal registry where the object will be stored (e.g., `action`, `type`), *value*, which is the key under which the object will be registered, and object, the callable to be registered.

The following example shows how to register a custom type with a parser :

```
>>> import argparse
>>> parser = argparse.ArgumentParser()
>>> parser.register('type', 'hexadecimal integer', lambda s: int(s, 16))
>>> parser.add_argument('--foo', type='hexadecimal integer')
_StoreAction(option_strings=['--foo'], dest='foo', nargs=None, const=None,␣
↪default=None, type='hexadecimal integer', choices=None, required=False,␣
```

(suite sur la page suivante)

(suite de la page précédente)

```
↪help=None, metavar=None, deprecated=False)
>>> parser.parse_args(['--foo', '0xFA'])
Namespace(foo=250)
>>> parser.parse_args(['--foo', '1.2'])
usage: PROG [-h] [--foo FOO]
PROG: error: argument --foo: invalid 'hexadecimal integer' value: '1.2'
```

17.1.5 Exceptions

exception argparse.**ArgumentError**

An error from creating or using an argument (optional or positional).

The string value of this exception is the message, augmented with information about the argument that caused it.

exception argparse.**ArgumentTypeError**

Raised when something goes wrong converting a command line string to a type.

Guides et tutoriels

Tutoriel *argparse*

auteur
Tshepang Mbambo

Ce tutoriel est destiné à être une introduction en douceur à `argparse`, le module d'analyse de ligne de commande recommandé dans la bibliothèque standard de Python.

> **Note**
>
> The standard library includes two other libraries directly related to command-line parameter processing : the lower level `optparse` module (which may require more code to configure for a given application, but also allows an application to request behaviors that `argparse` doesn't support), and the very low level `getopt` (which specifically serves as an equivalent to the `getopt()` family of functions available to C programmers). While neither of those modules is covered directly in this guide, many of the core concepts in `argparse` first originated in `optparse`, so some aspects of this tutorial will also be relevant to `optparse` users.

Concepts

Commençons par l'utilisation de la commande **`ls`** pour voir le type de fonctionnalité que nous allons étudier dans ce tutoriel d'introduction :

```
$ ls
cpython  devguide  prog.py  pypy  rm-unused-function.patch
$ ls pypy
ctypes_configure  demo  dotviewer  include  lib_pypy  lib-python ...
$ ls -l
total 20
drwxr-xr-x 19 wena wena 4096 Feb 18 18:51 cpython
drwxr-xr-x  4 wena wena 4096 Feb  8 12:04 devguide
-rwxr-xr-x  1 wena wena  535 Feb 19 00:05 prog.py
drwxr-xr-x 14 wena wena 4096 Feb  7 00:59 pypy
-rw-r--r--  1 wena wena  741 Feb 18 01:01 rm-unused-function.patch
$ ls --help
Usage: ls [OPTION]... [FILE]...
List information about the FILEs (the current directory by default).
```

(suite sur la page suivante)

(suite de la page précédente)

```
Sort entries alphabetically if none of -cftuvSUX nor --sort is specified.
...
```

Quelques concepts que l'on peut apprendre avec les quatre commandes :

— La commande **ls** est utile quand elle est exécutée sans aucun paramètre. Elle affiche par défaut le contenu du dossier courant.

— Si l'on veut plus que ce qui est proposé par défaut, il faut l'indiquer. Dans le cas présent, on veut afficher un dossier différent : `pypy`. Ce que l'on a fait c'est spécifier un argument positionnel. C'est appelé ainsi car cela permet au programme de savoir quoi faire avec la valeur en se basant seulement sur sa position dans la ligne de commande. Ce concept est plus pertinent pour une commande comme **cp** dont l'usage de base est `cp SRC DEST`. Le premier argument est *ce que vous voulez copier* et le second est *où vous voulez le copier*.

— Maintenant, supposons que l'on veuille changer la façon dont le programme agit. Dans notre exemple, on affiche plus d'information pour chaque ficher que simplement leur nom. Dans ce cas, `-l` est un argument facultatif.

— C'est un fragment du texte d'aide. Cela peut être très utile quand on tombe sur un programme que l'on à jamais utilisé auparavant car on peut comprendre son fonctionnement simplement en lisant l'aide associée.

Les bases

Commençons par un exemple très simple qui ne fait (quasiment) rien :

```
import argparse
parser = argparse.ArgumentParser()
parser.parse_args()
```

Ce qui suit est le résultat de l'exécution du code :

```
$ python prog.py
$ python prog.py --help
usage: prog.py [-h]

options:
  -h, --help  show this help message and exit
$ python prog.py --verbose
usage: prog.py [-h]
prog.py: error: unrecognized arguments: --verbose
$ python prog.py foo
usage: prog.py [-h]
prog.py: error: unrecognized arguments: foo
```

Voilà ce qu'il se passe :

— Exécuter le script sans aucun paramètre a pour effet de ne rien afficher sur la sortie d'erreur. Ce n'est pas très utile.

— Le deuxième commence à montrer l'intérêt du module *argparse*. On n'a quasiment rien fait mais on a déjà un beau message d'aide.

— L'option `--help`, que l'on peut aussi raccourcir en `-h`, est la seule option que l'on a gratuitement (pas besoin de la préciser). Préciser quoi que ce soit d'autre entraîne une erreur. Mais même dans ce cas, on reçoit aussi un message utile, toujours gratuitement.

Introduction aux arguments positionnels

Un exemple :

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("echo")
```

(suite sur la page suivante)

(suite de la page précédente)

```
args = parser.parse_args()
print(args.echo)
```

On exécute le code :

```
$ python prog.py
usage: prog.py [-h] echo
prog.py: error: the following arguments are required: echo
$ python prog.py --help
usage: prog.py [-h] echo

positional arguments:
  echo

options:
  -h, --help  show this help message and exit
$ python prog.py foo
foo
```

Voilà ce qu'il se passe :

- We've added the *add_argument()* method, which is what we use to specify which command-line options the program is willing to accept. In this case, I've named it `echo` so that it's in line with its function.
- Utiliser le programme nécessite maintenant que l'on précise un paramètre.
- The *parse_args()* method actually returns some data from the options specified, in this case, `echo`.
- La variable est comme une forme de 'magie' que *argparse* effectue gratuitement (c.-à-d. pas besoin de préciser dans quelle variable la valeur est stockée). Vous aurez aussi remarqué que le nom est le même que l'argument en chaîne de caractères donné à la méthode : `echo`.

Notez cependant que, même si l'affichage d'aide paraît bien , il n'est pas aussi utile qu'il pourrait l'être. Par exemple, on peut lire que `echo` est un argument positionnel mais on ne peut pas savoir ce que cela fait autrement qu'en le devinant ou en lisant le code source. Donc, rendons-le un peu plus utile :

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("echo", help="echo the string you use here")
args = parser.parse_args()
print(args.echo)
```

Et on obtient :

```
$ python prog.py -h
usage: prog.py [-h] echo

positional arguments:
  echo        echo the string you use here

options:
  -h, --help  show this help message and exit
```

À présent, que diriez-vous de faire quelque chose d'encore plus utile :

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", help="display a square of a given number")
args = parser.parse_args()
print(args.square**2)
```

Ce qui suit est le résultat de l'exécution du code :

```
$ python prog.py 4
Traceback (most recent call last):
  File "prog.py", line 5, in <module>
    print(args.square**2)
TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'int'
```

Cela n'a pas très bien fonctionné. C'est parce que `argparse` traite les paramètres que l'on donne comme des chaînes de caractères à moins qu'on ne lui indique de faire autrement. Donc, disons à `argparse` de traiter cette entrée comme un entier :

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", help="display a square of a given number",
                    type=int)
args = parser.parse_args()
print(args.square**2)
```

Ce qui suit est le résultat de l'exécution du code :

```
$ python prog.py 4
16
$ python prog.py four
usage: prog.py [-h] square
prog.py: error: argument square: invalid int value: 'four'
```

Cela a bien fonctionné. Maintenant le programme va même s'arrêter si l'entrée n'est pas légale avant de procéder à l'exécution.

Introduction aux arguments optionnels

Jusqu'à maintenant, on a joué avec les arguments positionnels. Regardons comment ajouter des paramètres optionnels :

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("--verbosity", help="increase output verbosity")
args = parser.parse_args()
if args.verbosity:
    print("verbosity turned on")
```

Et le résultat :

```
$ python prog.py --verbosity 1
verbosity turned on
$ python prog.py
$ python prog.py --help
usage: prog.py [-h] [--verbosity VERBOSITY]

options:
  -h, --help            show this help message and exit
  --verbosity VERBOSITY
                        increase output verbosity
$ python prog.py --verbosity
usage: prog.py [-h] [--verbosity VERBOSITY]
prog.py: error: argument --verbosity: expected one argument
```

Voilà ce qu'il se passe :

— Le programme est écrit de sorte qu'il n'affiche rien sauf si l'option `--verbosity` est précisée.

— To show that the option is actually optional, there is no error when running the program without it. Note that by default, if an optional argument isn't used, the relevant variable, in this case `args.verbosity`, is given `None` as a value, which is the reason it fails the truth test of the `if` statement.
— Le message d'aide est quelque peu différent.
— Quand on utilise l'option `--verbosity` on doit aussi préciser une valeur, n'importe laquelle.

L'exemple ci-dessus accepte des valeurs entières arbitraires pour `--verbosity` mais pour notre programme simple seule deux valeurs sont réellement utiles : `True` et `False`. Modifions le code en accord avec cela :

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("--verbose", help="increase output verbosity",
                    action="store_true")
args = parser.parse_args()
if args.verbose:
    print("verbosity turned on")
```

Et le résultat :

```
$ python prog.py --verbose
verbosity turned on
$ python prog.py --verbose 1
usage: prog.py [-h] [--verbose]
prog.py: error: unrecognized arguments: 1
$ python prog.py --help
usage: prog.py [-h] [--verbose]

options:
  -h, --help  show this help message and exit
  --verbose   increase output verbosity
```

Voilà ce qu'il se passe :

— The option is now more of a flag than something that requires a value. We even changed the name of the option to match that idea. Note that we now specify a new keyword, `action`, and give it the value `"store_true"`. This means that, if the option is specified, assign the value `True` to `args.verbose`. Not specifying it implies `False`.
— Dans l'esprit de ce que sont vraiment les options, pas des paramètres, il se plaint quand vous tentez de préciser une valeur.
— Notez que l'aide est différente.

Les paramètres raccourcis

Si vous êtes familier avec l'utilisation de la ligne de commande, vous avez dû remarquer que je n'ai pour l'instant rien dit au sujet des versions raccourcies des paramètres. C'est très simple :

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("-v", "--verbose", help="increase output verbosity",
                    action="store_true")
args = parser.parse_args()
if args.verbose:
    print("verbosity turned on")
```

Et voilà :

```
$ python prog.py -v
verbosity turned on
$ python prog.py --help
usage: prog.py [-h] [-v]
```

(suite sur la page suivante)

(suite de la page précédente)

```
options:
  -h, --help     show this help message and exit
  -v, --verbose  increase output verbosity
```

Notez que la nouvelle option est aussi indiquée dans l'aide.

Combinaison d'arguments positionnels et optionnels

Notre programme continue de croître en complexité :

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int,
                    help="display a square of a given number")
parser.add_argument("-v", "--verbose", action="store_true",
                    help="increase output verbosity")
args = parser.parse_args()
answer = args.square**2
if args.verbose:
    print(f"the square of {args.square} equals {answer}")
else:
    print(answer)
```

Et voilà le résultat :

```
$ python prog.py
usage: prog.py [-h] [-v] square
prog.py: error: the following arguments are required: square
$ python prog.py 4
16
$ python prog.py 4 --verbose
the square of 4 equals 16
$ python prog.py --verbose 4
the square of 4 equals 16
```

— Nous avons ajouté un argument nommé, d'où le message d'erreur.
— Notez que l'ordre importe peu.

Qu'en est-il si nous donnons à ce programme la possibilité d'avoir plusieurs niveaux de verbosité, et que celui-ci les prend en compte :

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int,
                    help="display a square of a given number")
parser.add_argument("-v", "--verbosity", type=int,
                    help="increase output verbosity")
args = parser.parse_args()
answer = args.square**2
if args.verbosity == 2:
    print(f"the square of {args.square} equals {answer}")
elif args.verbosity == 1:
    print(f"{args.square}^2 == {answer}")
else:
    print(answer)
```

Et le résultat :

```
$ python prog.py 4
16
$ python prog.py 4 -v
usage: prog.py [-h] [-v VERBOSITY] square
prog.py: error: argument -v/--verbosity: expected one argument
$ python prog.py 4 -v 1
4^2 == 16
$ python prog.py 4 -v 2
the square of 4 equals 16
$ python prog.py 4 -v 3
16
```

Tout semble bon sauf le dernier, qui montre que notre programme contient un bogue. Corrigeons cela en restreignant les valeurs que `--verbosity` accepte :

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int,
                    help="display a square of a given number")
parser.add_argument("-v", "--verbosity", type=int, choices=[0, 1, 2],
                    help="increase output verbosity")
args = parser.parse_args()
answer = args.square**2
if args.verbosity == 2:
    print(f"the square of {args.square} equals {answer}")
elif args.verbosity == 1:
    print(f"{args.square}^2 == {answer}")
else:
    print(answer)
```

Et le résultat :

```
$ python prog.py 4 -v 3
usage: prog.py [-h] [-v {0,1,2}] square
prog.py: error: argument -v/--verbosity: invalid choice: 3 (choose from 0, 1, 2)
$ python prog.py 4 -h
usage: prog.py [-h] [-v {0,1,2}] square

positional arguments:
  square                display a square of a given number

options:
  -h, --help            show this help message and exit
  -v, --verbosity {0,1,2}
                        increase output verbosity
```

Notez que ce changement est pris en compte à la fois dans le message d'erreur et dans le texte d'aide.

Essayons maintenant une approche différente pour jouer sur la verbosité, ce qui arrive fréquemment. Cela correspond également à comment le programme CPython gère ses propres paramètres de verbosité (jetez un œil sur la sortie de la commande `python --help`) :

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int,
                    help="display the square of a given number")
parser.add_argument("-v", "--verbosity", action="count",
                    help="increase output verbosity")
```

(suite sur la page suivante)

(suite de la page précédente)

```
args = parser.parse_args()
answer = args.square**2
if args.verbosity == 2:
    print(f"the square of {args.square} equals {answer}")
elif args.verbosity == 1:
    print(f"{args.square}^2 == {answer}")
else:
    print(answer)
```

Nous avons introduit une autre action, `"count"`, pour compter le nombre d'occurrences d'une option en particulier :

```
$ python prog.py 4
16
$ python prog.py 4 -v
4^2 == 16
$ python prog.py 4 -vv
the square of 4 equals 16
$ python prog.py 4 --verbosity --verbosity
the square of 4 equals 16
$ python prog.py 4 -v 1
usage: prog.py [-h] [-v] square
prog.py: error: unrecognized arguments: 1
$ python prog.py 4 -h
usage: prog.py [-h] [-v] square

positional arguments:
  square           display a square of a given number

options:
  -h, --help       show this help message and exit
  -v, --verbosity  increase output verbosity
$ python prog.py 4 -vvv
16
```

- Oui, c'est maintenant d'avantage une option (similaire à `action="store_true"`) de la version précédente de notre script. Cela devrait expliquer le message d'erreur.
- Cela se comporte de la même manière que l'action `"store_true"`.
- Maintenant voici une démonstration de ce que l'action `"count"` fait. Vous avez sûrement vu ce genre d'utilisation auparavant.
- Et si vous ne spécifiez pas l'option `-v`, cette option prendra la valeur `None`.
- Comme on s'y attend, en spécifiant l'option dans sa forme longue, on devrait obtenir la même sortie.
- Malheureusement, notre sortie d'aide n'est pas très informative à propos des nouvelles possibilités de notre programme, mais cela peut toujours être corrigé en améliorant sa documentation (en utilisant l'argument `help`).
- La dernière sortie du programme montre que celui-ci contient un bogue.

Corrigeons :

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int,
                    help="display a square of a given number")
parser.add_argument("-v", "--verbosity", action="count",
                    help="increase output verbosity")
args = parser.parse_args()
answer = args.square**2
```

(suite sur la page suivante)

(suite de la page précédente)

```
# bugfix: replace == with >=
if args.verbosity >= 2:
    print(f"the square of {args.square} equals {answer}")
elif args.verbosity >= 1:
    print(f"{args.square}^2 == {answer}")
else:
    print(answer)
```

Et c'est ce que ça donne :

```
$ python prog.py 4 -vvv
the square of 4 equals 16
$ python prog.py 4 -vvvv
the square of 4 equals 16
$ python prog.py 4
Traceback (most recent call last):
  File "prog.py", line 11, in <module>
    if args.verbosity >= 2:
TypeError: '>=' not supported between instances of 'NoneType' and 'int'
```

— Les premières exécutions du programme sont correctes, et le bogue que nous avons eu est corrigé. Cela dit, nous voulons que n'importe quelle valeur >= 2 rende le programme aussi verbeux que possible.
— La troisième sortie de programme n'est pas si bien que ça.

Corrigeons ce bogue :

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int,
                    help="display a square of a given number")
parser.add_argument("-v", "--verbosity", action="count", default=0,
                    help="increase output verbosity")
args = parser.parse_args()
answer = args.square**2
if args.verbosity >= 2:
    print(f"the square of {args.square} equals {answer}")
elif args.verbosity >= 1:
    print(f"{args.square}^2 == {answer}")
else:
    print(answer)
```

Nous venons juste d'introduire un nouveau mot clef, `default`. Nous l'avons définit à `0` pour le rendre comparable aux autres valeurs. Rappelez-vous que par défaut, si un argument optionnel n'est pas spécifié, il sera définit à `None`, et ne pourra pas être comparé à une valeur de type entier (une erreur *`TypeError`* serait alors levée).

Et :

```
$ python prog.py 4
16
```

Vous pouvez aller assez loin seulement avec ce que nous avons appris jusqu'à maintenant, et nous n'avons qu'aperçu la surface. Le module *`argparse`* est très puissant, et nous allons l'explorer un peu plus avant la fin de ce tutoriel.

Aller un peu plus loin

Qu'en est-il si nous souhaitons étendre notre mini programme pour le rendre capable de calculer d'autres puissances, et pas seulement des carrés :

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("x", type=int, help="the base")
parser.add_argument("y", type=int, help="the exponent")
parser.add_argument("-v", "--verbosity", action="count", default=0)
args = parser.parse_args()
answer = args.x**args.y
if args.verbosity >= 2:
    print(f"{args.x} to the power {args.y} equals {answer}")
elif args.verbosity >= 1:
    print(f"{args.x}^{args.y} == {answer}")
else:
    print(answer)
```

Sortie :

```
$ python prog.py
usage: prog.py [-h] [-v] x y
prog.py: error: the following arguments are required: x, y
$ python prog.py -h
usage: prog.py [-h] [-v] x y

positional arguments:
  x                 the base
  y                 the exponent

options:
  -h, --help        show this help message and exit
  -v, --verbosity
$ python prog.py 4 2 -v
4^2 == 16
```

Il est à noter que jusqu'à présent nous avons utilisé le niveau de verbosité pour *changer* le texte qui est affiché. L'exemple suivant au contraire utilise le niveau de verbosité pour afficher *plus* de texte à la place :

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("x", type=int, help="the base")
parser.add_argument("y", type=int, help="the exponent")
parser.add_argument("-v", "--verbosity", action="count", default=0)
args = parser.parse_args()
answer = args.x**args.y
if args.verbosity >= 2:
    print(f"Running '{__file__}'")
if args.verbosity >= 1:
    print(f"{args.x}^{args.y} == ", end="")
print(answer)
```

Sortie :

```
$ python prog.py 4 2
16
$ python prog.py 4 2 -v
4^2 == 16
$ python prog.py 4 2 -vv
Running 'prog.py'
4^2 == 16
```

Specifying ambiguous arguments

When there is ambiguity in deciding whether an argument is positional or for an argument, `--` can be used to tell *parse_args()* that everything after that is a positional argument :

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-n', nargs='+')
>>> parser.add_argument('args', nargs='*')

>>> # ambiguous, so parse_args assumes it's an option
>>> parser.parse_args(['-f'])
usage: PROG [-h] [-n N [N ...]] [args ...]
PROG: error: unrecognized arguments: -f

>>> parser.parse_args(['--', '-f'])
Namespace(args=['-f'], n=None)

>>> # ambiguous, so the -n option greedily accepts arguments
>>> parser.parse_args(['-n', '1', '2', '3'])
Namespace(args=[], n=['1', '2', '3'])

>>> parser.parse_args(['-n', '1', '--', '2', '3'])
Namespace(args=['2', '3'], n=['1'])
```

Paramètres en conflit

So far, we have been working with two methods of an *argparse.ArgumentParser* instance. Let's introduce a third one, *add_mutually_exclusive_group()*. It allows for us to specify options that conflict with each other. Let's also change the rest of the program so that the new functionality makes more sense : we'll introduce the `--quiet` option, which will be the opposite of the `--verbose` one :

```
import argparse

parser = argparse.ArgumentParser()
group = parser.add_mutually_exclusive_group()
group.add_argument("-v", "--verbose", action="store_true")
group.add_argument("-q", "--quiet", action="store_true")
parser.add_argument("x", type=int, help="the base")
parser.add_argument("y", type=int, help="the exponent")
args = parser.parse_args()
answer = args.x**args.y

if args.quiet:
    print(answer)
elif args.verbose:
    print(f"{args.x} to the power {args.y} equals {answer}")
else:
    print(f"{args.x}^{args.y} == {answer}")
```

Notre programme est maintenant plus simple, et nous avons perdu des fonctionnalités pour faire cette démonstration. Peu importe, voici la sortie du programme :

```
$ python prog.py 4 2
4^2 == 16
$ python prog.py 4 2 -q
16
$ python prog.py 4 2 -v
4 to the power 2 equals 16
```

(suite sur la page suivante)

(suite de la page précédente)

```
$ python prog.py 4 2 -vq
usage: prog.py [-h] [-v | -q] x y
prog.py: error: argument -q/--quiet: not allowed with argument -v/--verbose
$ python prog.py 4 2 -v --quiet
usage: prog.py [-h] [-v | -q] x y
prog.py: error: argument -q/--quiet: not allowed with argument -v/--verbose
```

Cela devrait être facile à suivre. J'ai ajouté cette dernière sortie pour que vous puissiez voir le genre de flexibilité que vous pouvez avoir, par exemple pour faire un mélange entre des paramètres courts et longs.

Avant d'en finir, vous voudrez certainement dire à vos utilisateurs quel est le but principal de votre programme, juste dans le cas ou ils ne le sauraient pas :

```
import argparse

parser = argparse.ArgumentParser(description="calculate X to the power of Y")
group = parser.add_mutually_exclusive_group()
group.add_argument("-v", "--verbose", action="store_true")
group.add_argument("-q", "--quiet", action="store_true")
parser.add_argument("x", type=int, help="the base")
parser.add_argument("y", type=int, help="the exponent")
args = parser.parse_args()
answer = args.x**args.y

if args.quiet:
    print(answer)
elif args.verbose:
    print(f"{args.x} to the power {args.y} equals {answer}")
else:
    print(f"{args.x}^{args.y} == {answer}")
```

Notez cette nuance dans le texte d'utilisation. Les options `[-v | -q]` nous disent que nous pouvons utiliser au choix `-v` ou `-q`, mais pas les deux ensemble :

```
$ python prog.py --help
usage: prog.py [-h] [-v | -q] x y

calculate X to the power of Y

positional arguments:
  x              the base
  y              the exponent

options:
  -h, --help     show this help message and exit
  -v, --verbose
  -q, --quiet
```

How to translate the argparse output

The output of the `argparse` module such as its help text and error messages are all made translatable using the `gettext` module. This allows applications to easily localize messages produced by `argparse`. See also *Internationaliser vos programmes et modules*.

For instance, in this `argparse` output :

```
$ python prog.py --help
usage: prog.py [-h] [-v | -q] x y

calculate X to the power of Y

positional arguments:
  x              the base
  y              the exponent

options:
  -h, --help     show this help message and exit
  -v, --verbose
  -q, --quiet
```

The strings `usage:`, `positional arguments:`, `options:` and `show this help message and exit` are all translatable.

In order to translate these strings, they must first be extracted into a `.po` file. For example, using Babel, run this command :

```
$ pybabel extract -o messages.po /usr/lib/python3.12/argparse.py
```

This command will extract all translatable strings from the *argparse* module and output them into a file named `messages.po`. This command assumes that your Python installation is in `/usr/lib`.

You can find out the location of the *argparse* module on your system using this script :

```
import argparse
print(argparse.__file__)
```

Once the messages in the `.po` file are translated and the translations are installed using *gettext*, *argparse* will be able to display the translated messages.

To translate your own strings in the *argparse* output, use *gettext*.

Custom type converters

The *argparse* module allows you to specify custom type converters for your command-line arguments. This allows you to modify user input before it's stored in the *argparse.Namespace*. This can be useful when you need to pre-process the input before it is used in your program.

When using a custom type converter, you can use any callable that takes a single string argument (the argument value) and returns the converted value. However, if you need to handle more complex scenarios, you can use a custom action class with the **action** parameter instead.

For example, let's say you want to handle arguments with different prefixes and process them accordingly :

```
import argparse

parser = argparse.ArgumentParser(prefix_chars='-+')

parser.add_argument('-a', metavar='<value>', action='append',
                    type=lambda x: ('-', x))
parser.add_argument('+a', metavar='<value>', action='append',
                    type=lambda x: ('+', x))

args = parser.parse_args()
print(args)
```

Sortie :

```
$ python prog.py -a value1 +a value2
Namespace(a=[('-', 'value1'), ('+', 'value2')])
```

In this example, we :

- Created a parser with custom prefix characters using the `prefix_chars` parameter.
- Defined two arguments, `-a` and `+a`, which used the `type` parameter to create custom type converters to store the value in a tuple with the prefix.

Without the custom type converters, the arguments would have treated the `-a` and `+a` as the same argument, which would have been undesirable. By using custom type converters, we were able to differentiate between the two arguments.

Conclusion

Le module *argparse* offre bien plus que ce qui est montré ici. Sa documentation est assez détaillée, complète et pleine d'exemples. En ayant accompli ce tutoriel, vous pourrez facilement comprendre cette documentation sans vous sentir dépassé.

Migrating `optparse` code to `argparse`

The *argparse* module offers several higher level features not natively provided by the *optparse* module, including :

- gère les arguments positionnels ;
- Supporting subcommands.
- permet d'utiliser les alternatives `+` ou `/` comme préfixes d'option ;
- prend en charge la répétition de valeurs (zéro ou plus, un ou plus) ;
- fournit des messages d'aide plus complets ;
- fournit une interface plus simple pour les types et les actions personnalisés.

Originally, the *argparse* module attempted to maintain compatibility with *optparse*. However, the fundamental design differences between supporting declarative command line option processing (while leaving positional argument processing to application code), and supporting both named options and positional arguments in the declarative interface mean that the API has diverged from that of `optparse` over time.

As described in *Choosing an argument parsing library*, applications that are currently using *optparse* and are happy with the way it works can just continue to use `optparse`.

Application developers that are considering migrating should also review the list of intrinsic behavioural differences described in that section before deciding whether or not migration is desirable.

For applications that do choose to migrate from *optparse* to *argparse*, the following suggestions should be helpful :

- remplacer tous les appels à *optparse.OptionParser.add_option()* par des appels à *ArgumentParser.add_argument()* ;
- remplacer `(options, args) = parser.parse_args()` par `args = parser.parse_args()` et ajouter des appels à *ArgumentParser.add_argument()* pour les arguments positionnels. Prenez note que les valeurs précédemment appelées `options` sont appelées `args` dans le contexte d'*argparse* ;
- remplacer *optparse.OptionParser.disable_interspersed_args()* en appelant *parse_intermixed_args()* plutôt que *parse_args()* ;
- remplacer les actions de rappel (*callback actions* en anglais) et les arguments nommés `callback_*` par des arguments `type` et `actions` ;
- remplacer les chaînes représentant le nom des types pour l'argument nommé `type` par les objets types correspondants (par exemple : `int`, `float`, `complex`, etc) ;
- remplacer *optparse.Values* par *Namespace* ; et *optparse.OptionError* et *optparse.OptionValueError* par *ArgumentError* ;
- remplacer les chaînes avec des arguments de formatage implicite (tels que `%default` ou `%prog`) par la syntaxe standard de Python pour l'interpolation d'un dictionnaire dans les chaînes de formatage (c'est-à-dire `%(default)s` et `%(prog)s`) ;
- remplacer l'argument `version` du constructeur d'`OptionParser` par un appel à `parser.add_argument('--version', action='version', version='<la version>')`.

17.2 optparse --- Parser for command line options

Code source : Lib/optparse.py

17.2.1 Choosing an argument parsing library

The standard library includes three argument parsing libraries :

- *getopt* : a module that closely mirrors the procedural C `getopt` API. Included in the standard library since before the initial Python 1.0 release.
- *optparse* : a declarative replacement for `getopt` that provides equivalent functionality without requiring each application to implement its own procedural option parsing logic. Included in the standard library since the Python 2.3 release.
- *argparse* : a more opinionated alternative to `optparse` that provides more functionality by default, at the expense of reduced application flexibility in controlling exactly how arguments are processed. Included in the standard library since the Python 2.7 and Python 3.2 releases.

In the absence of more specific argument parsing design constraints, *argparse* is the recommended choice for implementing command line applications, as it offers the highest level of baseline functionality with the least application level code.

getopt is retained almost entirely for backwards compatibility reasons. However, it also serves a niche use case as a tool for prototyping and testing command line argument handling in `getopt`-based C applications.

optparse should be considered as an alternative to *argparse* in the following cases :

- an application is already using *optparse* and doesn't want to risk the subtle behavioural changes that may arise when migrating to *argparse*
- the application requires additional control over the way options and positional parameters are interleaved on the command line (including the ability to disable the interleaving feature completely)
- the application requires additional control over the incremental parsing of command line elements (while `argparse` does support this, the exact way it works in practice is undesirable for some use cases)
- the application requires additional control over the handling of options which accept parameter values that may start with `-` (such as delegated options to be passed to invoked subprocesses)
- the application requires some other command line parameter processing behavior which `argparse` does not support, but which can be implemented in terms of the lower level interface offered by `optparse`

These considerations also mean that *optparse* is likely to provide a better foundation for library authors writing third party command line argument processing libraries.

As a concrete example, consider the following two command line argument parsing configurations, the first using `optparse`, and the second using `argparse` :

```
import optparse

if __name__ == '__main__':
    parser = optparse.OptionParser()
    parser.add_option('-o', '--output')
    parser.add_option('-v', dest='verbose', action='store_true')
    opts, args = parser.parse_args()
    process(args, output=opts.output, verbose=opts.verbose)
```

```
import argparse

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('-o', '--output')
    parser.add_argument('-v', dest='verbose', action='store_true')
    parser.add_argument('rest', nargs='*')
    args = parser.parse_args()
    process(args.rest, output=args.output, verbose=args.verbose)
```

The most obvious difference is that in the `optparse` version, the non-option arguments are processed separately by the application after the option processing is complete. In the `argparse` version, positional arguments are declared and processed in the same way as the named options.

However, the `argparse` version will also handle some parameter combination differently from the way the `optparse` version would handle them. For example (amongst other differences) :

- supplying `-o -v` gives `output="-v"` and `verbose=False` when using `optparse`, but a usage error with `argparse` (complaining that no value has been supplied for `-o/--output`, since `-v` is interpreted as meaning the verbosity flag)
- similarly, supplying `-o --` gives `output="--"` and `args=()` when using `optparse`, but a usage error with `argparse` (also complaining that no value has been supplied for `-o/--output`, since `--` is interpreted as terminating the option processing and treating all remaining values as positional arguments)
- supplying `-o=foo` gives `output="=foo"` when using `optparse`, but gives `output="foo"` with `argparse` (since = is special cased as an alternative separator for option parameter values)

Whether these differing behaviors in the `argparse` version are considered desirable or a problem will depend on the specific command line application use case.

Voir aussi

click is a third party argument processing library (originally based on `optparse`), which allows command line applications to be developed as a set of decorated command implementation functions.

Other third party libraries, such as typer or msgspec-click, allow command line interfaces to be specified in ways that more effectively integrate with static checking of Python type annotations.

17.2.2 Introduction

optparse is a more convenient, flexible, and powerful library for parsing command-line options than the minimalist *getopt* module. *optparse* uses a more declarative style of command-line parsing : you create an instance of *OptionParser*, populate it with options, and parse the command line. *optparse* allows users to specify options in the conventional GNU/POSIX syntax, and additionally generates usage and help messages for you.

Here's an example of using *optparse* in a simple script :

```
from optparse import OptionParser
...
parser = OptionParser()
parser.add_option("-f", "--file", dest="filename",
                  help="write report to FILE", metavar="FILE")
parser.add_option("-q", "--quiet",
                  action="store_false", dest="verbose", default=True,
                  help="don't print status messages to stdout")

(options, args) = parser.parse_args()
```

With these few lines of code, users of your script can now do the "usual thing" on the command-line, for example :

```
<yourscript> --file=outfile -q
```

As it parses the command line, *optparse* sets attributes of the `options` object returned by *parse_args()* based on user-supplied command-line values. When *parse_args()* returns from parsing this command line, `options.filename` will be `"outfile"` and `options.verbose` will be `False`. *optparse* supports both long and short options, allows short options to be merged together, and allows options to be associated with their arguments in a variety of ways. Thus, the following command lines are all equivalent to the above example :

```
<yourscript> -f outfile --quiet
<yourscript> --quiet --file outfile
<yourscript> -q -foutfile
<yourscript> -qfoutfile
```

Additionally, users can run one of the following

```
<yourscript> -h
<yourscript> --help
```

and *optparse* will print out a brief summary of your script's options :

```
Usage: <yourscript> [options]

Options:
  -h, --help            show this help message and exit
  -f FILE, --file=FILE  write report to FILE
  -q, --quiet           don't print status messages to stdout
```

where the value of *yourscript* is determined at runtime (normally from `sys.argv[0]`).

17.2.3 Background

optparse was explicitly designed to encourage the creation of programs with straightforward command-line interfaces that follow the conventions established by the `getopt()` family of functions available to C developers. To that end, it supports only the most common command-line syntax and semantics conventionally used under Unix. If you are unfamiliar with these conventions, reading this section will allow you to acquaint yourself with them.

Terminology

argument

a string entered on the command-line, and passed by the shell to `execl()` or `execv()`. In Python, arguments are elements of `sys.argv[1:]` (`sys.argv[0]` is the name of the program being executed). Unix shells also use the term "word".

It is occasionally desirable to substitute an argument list other than `sys.argv[1:]`, so you should read "argument" as "an element of `sys.argv[1:]`, or of some other list provided as a substitute for `sys.argv[1:]`".

option

an argument used to supply extra information to guide or customize the execution of a program. There are many different syntaxes for options ; the traditional Unix syntax is a hyphen ("-") followed by a single letter, e.g. `-x` or `-F`. Also, traditional Unix syntax allows multiple options to be merged into a single argument, e.g. `-x -F` is equivalent to `-xF`. The GNU project introduced `--` followed by a series of hyphen-separated words, e.g. `--file` or `--dry-run`. These are the only two option syntaxes provided by *optparse*.

Some other option syntaxes that the world has seen include :

— a hyphen followed by a few letters, e.g. `-pf` (this is *not* the same as multiple options merged into a single argument)

— a hyphen followed by a whole word, e.g. `-file` (this is technically equivalent to the previous syntax, but they aren't usually seen in the same program)

— a plus sign followed by a single letter, or a few letters, or a word, e.g. `+f`, `+rgb`

— a slash followed by a letter, or a few letters, or a word, e.g. `/f`, `/file`

These option syntaxes are not supported by *optparse*, and they never will be. This is deliberate : the first three are non-standard on any environment, and the last only makes sense if you're exclusively targeting Windows or certain legacy platforms (e.g. VMS, MS-DOS).

option argument

an argument that follows an option, is closely associated with that option, and is consumed from the argument list when that option is. With *optparse*, option arguments may either be in a separate argument from their option :

```
-f foo
--file foo
```

or included in the same argument :

```
-ffoo
--file=foo
```

Typically, a given option either takes an argument or it doesn't. Lots of people want an "optional option arguments" feature, meaning that some options will take an argument if they see it, and won't if they don't. This is somewhat controversial, because it makes parsing ambiguous : if `-a` takes an optional argument and `-b` is another option entirely, how do we interpret `-ab` ? Because of this ambiguity, *optparse* does not support this feature.

argument positionnel
: something leftover in the argument list after options have been parsed, i.e. after options and their arguments have been parsed and removed from the argument list.

required option
: an option that must be supplied on the command-line ; note that the phrase "required option" is self-contradictory in English. *optparse* doesn't prevent you from implementing required options, but doesn't give you much help at it either.

For example, consider this hypothetical command-line :

```
prog -v --report report.txt foo bar
```

`-v` and `--report` are both options. Assuming that `--report` takes one argument, `report.txt` is an option argument. `foo` and `bar` are positional arguments.

What are options for ?

Options are used to provide extra information to tune or customize the execution of a program. In case it wasn't clear, options are usually *optional*. A program should be able to run just fine with no options whatsoever. (Pick a random program from the Unix or GNU toolsets. Can it run without any options at all and still make sense ? The main exceptions are `find`, `tar`, and `dd`---all of which are mutant oddballs that have been rightly criticized for their non-standard syntax and confusing interfaces.)

Lots of people want their programs to have "required options". Think about it. If it's required, then it's *not optional* ! If there is a piece of information that your program absolutely requires in order to run successfully, that's what positional arguments are for.

As an example of good command-line interface design, consider the humble `cp` utility, for copying files. It doesn't make much sense to try to copy files without supplying a destination and at least one source. Hence, `cp` fails if you run it with no arguments. However, it has a flexible, useful syntax that does not require any options at all :

```
cp SOURCE DEST
cp SOURCE ... DEST-DIR
```

You can get pretty far with just that. Most `cp` implementations provide a bunch of options to tweak exactly how the files are copied : you can preserve mode and modification time, avoid following symlinks, ask before clobbering existing files, etc. But none of this distracts from the core mission of `cp`, which is to copy either one file to another, or several files to another directory.

What are positional arguments for ?

Positional arguments are for those pieces of information that your program absolutely, positively requires to run.

A good user interface should have as few absolute requirements as possible. If your program requires 17 distinct pieces of information in order to run successfully, it doesn't much matter *how* you get that information from the user---most people will give up and walk away before they successfully run the program. This applies whether the user interface is a command-line, a configuration file, or a GUI : if you make that many demands on your users, most of them will simply give up.

In short, try to minimize the amount of information that users are absolutely required to supply---use sensible defaults whenever possible. Of course, you also want to make your programs reasonably flexible. That's what options are for.

Again, it doesn't matter if they are entries in a config file, widgets in the "Preferences" dialog of a GUI, or command-line options---the more options you implement, the more flexible your program is, and the more complicated its implementation becomes. Too much flexibility has drawbacks as well, of course ; too many options can overwhelm users and make your code much harder to maintain.

17.2.4 Tutoriel

While *optparse* is quite flexible and powerful, it's also straightforward to use in most cases. This section covers the code patterns that are common to any *optparse*-based program.

First, you need to import the OptionParser class ; then, early in the main program, create an OptionParser instance :

```
from optparse import OptionParser
...
parser = OptionParser()
```

Then you can start defining options. The basic syntax is :

```
parser.add_option(opt_str, ...,
                  attr=value, ...)
```

Each option has one or more option strings, such as `-f` or `--file`, and several option attributes that tell *optparse* what to expect and what to do when it encounters that option on the command line.

Typically, each option will have one short option string and one long option string, e.g. :

```
parser.add_option("-f", "--file", ...)
```

You're free to define as many short option strings and as many long option strings as you like (including zero), as long as there is at least one option string overall.

The option strings passed to *OptionParser.add_option()* are effectively labels for the option defined by that call. For brevity, we will frequently refer to *encountering an option* on the command line ; in reality, *optparse* encounters *option strings* and looks up options from them.

Once all of your options are defined, instruct *optparse* to parse your program's command line :

```
(options, args) = parser.parse_args()
```

(If you like, you can pass a custom argument list to *parse_args()*, but that's rarely necessary : by default it uses `sys.argv[1:]`.)

parse_args() returns two values :

- `options`, an object containing values for all of your options---e.g. if `--file` takes a single string argument, then `options.file` will be the filename supplied by the user, or `None` if the user did not supply that option
- `args`, the list of positional arguments leftover after parsing options

This tutorial section only covers the four most important option attributes : *action*, *type*, *dest* (destination), and *help*. Of these, *action* is the most fundamental.

Understanding option actions

Actions tell *optparse* what to do when it encounters an option on the command line. There is a fixed set of actions hard-coded into *optparse* ; adding new actions is an advanced topic covered in section *Extending optparse*. Most actions tell *optparse* to store a value in some variable---for example, take a string from the command line and store it in an attribute of `options`.

If you don't specify an option action, *optparse* defaults to `store`.

The store action

The most common option action is `store`, which tells *optparse* to take the next argument (or the remainder of the current argument), ensure that it is of the correct type, and store it to your chosen destination.

Par exemple :

```
parser.add_option("-f", "--file",
                  action="store", type="string", dest="filename")
```

Now let's make up a fake command line and ask *optparse* to parse it :

```
args = ["-f", "foo.txt"]
(options, args) = parser.parse_args(args)
```

When *optparse* sees the option string `-f`, it consumes the next argument, `foo.txt`, and stores it in `options.filename`. So, after this call to *parse_args()*, `options.filename` is `"foo.txt"`.

Some other option types supported by *optparse* are `int` and `float`. Here's an option that expects an integer argument :

```
parser.add_option("-n", type="int", dest="num")
```

Note that this option has no long option string, which is perfectly acceptable. Also, there's no explicit action, since the default is `store`.

Let's parse another fake command-line. This time, we'll jam the option argument right up against the option : since `-n42` (one argument) is equivalent to `-n 42` (two arguments), the code

```
(options, args) = parser.parse_args(["-n42"])
print(options.num)
```

affichera `42`.

If you don't specify a type, *optparse* assumes `string`. Combined with the fact that the default action is `store`, that means our first example can be a lot shorter :

```
parser.add_option("-f", "--file", dest="filename")
```

If you don't supply a destination, *optparse* figures out a sensible default from the option strings : if the first long option string is `--foo-bar`, then the default destination is `foo_bar`. If there are no long option strings, *optparse* looks at the first short option string : the default destination for `-f` is `f`.

optparse also includes the built-in `complex` type. Adding types is covered in section *Extending optparse*.

Handling boolean (flag) options

Flag options---set a variable to true or false when a particular option is seen---are quite common. *optparse* supports them with two separate actions, `store_true` and `store_false`. For example, you might have a `verbose` flag that is turned on with `-v` and off with `-q` :

```
parser.add_option("-v", action="store_true", dest="verbose")
parser.add_option("-q", action="store_false", dest="verbose")
```

Here we have two different options with the same destination, which is perfectly OK. (It just means you have to be a bit careful when setting default values---see below.)

When *optparse* encounters `-v` on the command line, it sets `options.verbose` to `True` ; when it encounters `-q`, `options.verbose` is set to `False`.

Other actions

Some other actions supported by *optparse* are :

"store_const"
: store a constant value, pre-set via *Option.const*

"append"
: append this option's argument to a list

"count"
: increment a counter by one

"callback"
: call a specified function

These are covered in section *Reference Guide*, and section *Option Callbacks*.

Valeurs par défaut

All of the above examples involve setting some variable (the "destination") when certain command-line options are seen. What happens if those options are never seen ? Since we didn't supply any defaults, they are all set to `None`. This is usually fine, but sometimes you want more control. *optparse* lets you supply a default value for each destination, which is assigned before the command line is parsed.

First, consider the verbose/quiet example. If we want *optparse* to set `verbose` to `True` unless `-q` is seen, then we can do this :

```
parser.add_option("-v", action="store_true", dest="verbose", default=True)
parser.add_option("-q", action="store_false", dest="verbose")
```

Since default values apply to the *destination* rather than to any particular option, and these two options happen to have the same destination, this is exactly equivalent :

```
parser.add_option("-v", action="store_true", dest="verbose")
parser.add_option("-q", action="store_false", dest="verbose", default=True)
```

Consider this :

```
parser.add_option("-v", action="store_true", dest="verbose", default=False)
parser.add_option("-q", action="store_false", dest="verbose", default=True)
```

Again, the default value for `verbose` will be `True` : the last default value supplied for any particular destination is the one that counts.

A clearer way to specify default values is the `set_defaults()` method of OptionParser, which you can call at any time before calling *parse_args()* :

```
parser.set_defaults(verbose=True)
parser.add_option(...)
(options, args) = parser.parse_args()
```

As before, the last value specified for a given option destination is the one that counts. For clarity, try to use one method or the other of setting default values, not both.

Generating help

optparse's ability to generate help and usage text automatically is useful for creating user-friendly command-line interfaces. All you have to do is supply a *help* value for each option, and optionally a short usage message for your whole program. Here's an OptionParser populated with user-friendly (documented) options :

```
usage = "usage: %prog [options] arg1 arg2"
parser = OptionParser(usage=usage)
parser.add_option("-v", "--verbose",
```

(suite sur la page suivante)

(suite de la page précédente)

```
                  action="store_true", dest="verbose", default=True,
                  help="make lots of noise [default]")
parser.add_option("-q", "--quiet",
                  action="store_false", dest="verbose",
                  help="be vewwy quiet (I'm hunting wabbits)")
parser.add_option("-f", "--filename",
                  metavar="FILE", help="write output to FILE")
parser.add_option("-m", "--mode",
                  default="intermediate",
                  help="interaction mode: novice, intermediate, "
                       "or expert [default: %default]")
```

If *optparse* encounters either `-h` or `--help` on the command-line, or if you just call `parser.print_help()`, it prints the following to standard output :

```
Usage: <yourscript> [options] arg1 arg2

Options:
  -h, --help            show this help message and exit
  -v, --verbose         make lots of noise [default]
  -q, --quiet           be vewwy quiet (I'm hunting wabbits)
  -f FILE, --filename=FILE
                        write output to FILE
  -m MODE, --mode=MODE  interaction mode: novice, intermediate, or
                        expert [default: intermediate]
```

(If the help output is triggered by a help option, *optparse* exits after printing the help text.)

There's a lot going on here to help *optparse* generate the best possible help message :

— the script defines its own usage message :

```
usage = "usage: %prog [options] arg1 arg2"
```

optparse expands `%prog` in the usage string to the name of the current program, i.e. `os.path.basename(sys.argv[0])`. The expanded string is then printed before the detailed option help.

If you don't supply a usage string, *optparse* uses a bland but sensible default : `"Usage: %prog [options]"`, which is fine if your script doesn't take any positional arguments.

— every option defines a help string, and doesn't worry about line-wrapping---*optparse* takes care of wrapping lines and making the help output look good.

— options that take a value indicate this fact in their automatically generated help message, e.g. for the "mode" option :

```
-m MODE, --mode=MODE
```

Here, "MODE" is called the meta-variable : it stands for the argument that the user is expected to supply to `-m`/`--mode`. By default, *optparse* converts the destination variable name to uppercase and uses that for the meta-variable. Sometimes, that's not what you want---for example, the `--filename` option explicitly sets `metavar="FILE"`, resulting in this automatically generated option description :

```
-f FILE, --filename=FILE
```

This is important for more than just saving space, though : the manually written help text uses the meta-variable `FILE` to clue the user in that there's a connection between the semi-formal syntax `-f FILE` and the informal semantic description "write output to FILE". This is a simple but effective way to make your help text a lot clearer and more useful for end users.

— options that have a default value can include `%default` in the help string---*optparse* will replace it with *str()* of the option's default value. If an option has no default value (or the default value is `None`), `%default` expands to `none`.

Grouping Options

When dealing with many options, it is convenient to group these options for better help output. An *OptionParser* can contain several option groups, each of which can contain several options.

An option group is obtained using the class *OptionGroup* :

class optparse.**OptionGroup**(*parser*, *title*, *description=None*)

where
— parser is the *OptionParser* instance the group will be inserted in to
— title is the group title
— description, optional, is a long description of the group

OptionGroup inherits from `OptionContainer` (like *OptionParser*) and so the `add_option()` method can be used to add an option to the group.

Once all the options are declared, using the *OptionParser* method `add_option_group()` the group is added to the previously defined parser.

Continuing with the parser defined in the previous section, adding an *OptionGroup* to a parser is easy :

```
group = OptionGroup(parser, "Dangerous Options",
                    "Caution: use these options at your own risk.  "
                    "It is believed that some of them bite.")
group.add_option("-g", action="store_true", help="Group option.")
parser.add_option_group(group)
```

This would result in the following help output :

```
Usage: <yourscript> [options] arg1 arg2

Options:
  -h, --help            show this help message and exit
  -v, --verbose         make lots of noise [default]
  -q, --quiet           be vewwy quiet (I'm hunting wabbits)
  -f FILE, --filename=FILE
                        write output to FILE
  -m MODE, --mode=MODE  interaction mode: novice, intermediate, or
                        expert [default: intermediate]

  Dangerous Options:
    Caution: use these options at your own risk.  It is believed that some
    of them bite.

    -g                  Group option.
```

A bit more complete example might involve using more than one group : still extending the previous example :

```
group = OptionGroup(parser, "Dangerous Options",
                    "Caution: use these options at your own risk.  "
                    "It is believed that some of them bite.")
group.add_option("-g", action="store_true", help="Group option.")
parser.add_option_group(group)

group = OptionGroup(parser, "Debug Options")
group.add_option("-d", "--debug", action="store_true",
                 help="Print debug information")
group.add_option("-s", "--sql", action="store_true",
                 help="Print all SQL statements executed")
group.add_option("-e", action="store_true", help="Print every action done")
parser.add_option_group(group)
```

that results in the following output :

```
Usage: <yourscript> [options] arg1 arg2

Options:
  -h, --help            show this help message and exit
  -v, --verbose         make lots of noise [default]
  -q, --quiet           be vewwy quiet (I'm hunting wabbits)
  -f FILE, --filename=FILE
                        write output to FILE
  -m MODE, --mode=MODE  interaction mode: novice, intermediate, or expert
                        [default: intermediate]

  Dangerous Options:
    Caution: use these options at your own risk.  It is believed that some
    of them bite.

    -g                  Group option.

  Debug Options:
    -d, --debug         Print debug information
    -s, --sql           Print all SQL statements executed
    -e                  Print every action done
```

Another interesting method, in particular when working programmatically with option groups is :

`OptionParser.`**`get_option_group`**(*opt_str*)

Return the `OptionGroup` to which the short or long option string *opt_str* (e.g. `'-o'` or `'--option'`) belongs. If there's no such `OptionGroup`, return `None`.

Printing a version string

Similar to the brief usage string, `optparse` can also print a version string for your program. You have to supply the string as the `version` argument to OptionParser :

```
parser = OptionParser(usage="%prog [-f] [-q]", version="%prog 1.0")
```

`%prog` is expanded just like it is in `usage`. Apart from that, `version` can contain anything you like. When you supply it, `optparse` automatically adds a `--version` option to your parser. If it encounters this option on the command line, it expands your `version` string (by replacing `%prog`), prints it to stdout, and exits.

For example, if your script is called `/usr/bin/foo` :

```
$ /usr/bin/foo --version
foo 1.0
```

The following two methods can be used to print and get the `version` string :

`OptionParser.`**`print_version`**(*file=None*)

Print the version message for the current program (`self.version`) to *file* (default stdout). As with `print_usage()`, any occurrence of `%prog` in `self.version` is replaced with the name of the current program. Does nothing if `self.version` is empty or undefined.

`OptionParser.`**`get_version`**()

Same as `print_version()` but returns the version string instead of printing it.

How optparse handles errors

There are two broad classes of errors that *optparse* has to worry about : programmer errors and user errors. Programmer errors are usually erroneous calls to *OptionParser.add_option()*, e.g. invalid option strings, unknown option attributes, missing option attributes, etc. These are dealt with in the usual way : raise an exception (either *optparse.OptionError* or *TypeError*) and let the program crash.

Handling user errors is much more important, since they are guaranteed to happen no matter how stable your code is. *optparse* can automatically detect some user errors, such as bad option arguments (passing `-n 4x` where `-n` takes an integer argument), missing arguments (`-n` at the end of the command line, where `-n` takes an argument of any type). Also, you can call `OptionParser.error()` to signal an application-defined error condition :

```
(options, args) = parser.parse_args()
...
if options.a and options.b:
    parser.error("options -a and -b are mutually exclusive")
```

In either case, *optparse* handles the error the same way : it prints the program's usage message and an error message to standard error and exits with error status 2.

Consider the first example above, where the user passes `4x` to an option that takes an integer :

```
$ /usr/bin/foo -n 4x
Usage: foo [options]

foo: error: option -n: invalid integer value: '4x'
```

Or, where the user fails to pass a value at all :

```
$ /usr/bin/foo -n
Usage: foo [options]

foo: error: -n option requires an argument
```

optparse-generated error messages take care always to mention the option involved in the error ; be sure to do the same when calling `OptionParser.error()` from your application code.

If *optparse*'s default error-handling behaviour does not suit your needs, you'll need to subclass OptionParser and override its `exit()` and/or `error()` methods.

Putting it all together

Here's what *optparse*-based scripts usually look like :

```
from optparse import OptionParser
...
def main():
    usage = "usage: %prog [options] arg"
    parser = OptionParser(usage)
    parser.add_option("-f", "--file", dest="filename",
                      help="read data from FILENAME")
    parser.add_option("-v", "--verbose",
                      action="store_true", dest="verbose")
    parser.add_option("-q", "--quiet",
                      action="store_false", dest="verbose")
    ...
    (options, args) = parser.parse_args()
    if len(args) != 1:
        parser.error("incorrect number of arguments")
    if options.verbose:
```

(suite sur la page suivante)

(suite de la page précédente)

```
        print("reading %s..." % options.filename)
    ...

if __name__ == "__main__":
    main()
```

17.2.5 Reference Guide

Creating the parser

The first step in using *optparse* is to create an OptionParser instance.

class optparse.**OptionParser**(...)

The OptionParser constructor has no required arguments, but a number of optional keyword arguments. You should always pass them as keyword arguments, i.e. do not rely on the order in which the arguments are declared.

usage (default : `"%prog [options]"`)
The usage summary to print when your program is run incorrectly or with a help option. When *optparse* prints the usage string, it expands `%prog` to `os.path.basename(sys.argv[0])` (or to `prog` if you passed that keyword argument). To suppress a usage message, pass the special value `optparse.SUPPRESS_USAGE`.

option_list (default : `[]`)
A list of Option objects to populate the parser with. The options in `option_list` are added after any options in `standard_option_list` (a class attribute that may be set by OptionParser subclasses), but before any version or help options. Deprecated ; use *add_option()* after creating the parser instead.

option_class (default : optparse.Option)
Class to use when adding options to the parser in *add_option()*.

version (default : `None`)
A version string to print when the user supplies a version option. If you supply a true value for `version`, *optparse* automatically adds a version option with the single option string `--version`. The substring `%prog` is expanded the same as for `usage`.

conflict_handler (default : `"error"`)
Specifies what to do when options with conflicting option strings are added to the parser ; see section *Conflicts between options*.

description (default : `None`)
A paragraph of text giving a brief overview of your program. *optparse* reformats this paragraph to fit the current terminal width and prints it when the user requests help (after `usage`, but before the list of options).

formatter (default : a new `IndentedHelpFormatter`)
An instance of optparse.HelpFormatter that will be used for printing help text. *optparse* provides two concrete classes for this purpose : IndentedHelpFormatter and TitledHelpFormatter.

add_help_option (default : `True`)
If true, *optparse* will add a help option (with option strings `-h` and `--help`) to the parser.

prog
The string to use when expanding `%prog` in `usage` and `version` instead of `os.path.basename(sys.argv[0])`.

epilog (default : `None`)
A paragraph of help text to print after the option help.

Populating the parser

There are several ways to populate the parser with options. The preferred way is by using *OptionParser.add_option()*, as shown in section *Tutoriel*. `add_option()` can be called in one of two ways :

— pass it an Option instance (as returned by `make_option()`)
— pass it any combination of positional and keyword arguments that are acceptable to `make_option()` (i.e., to the Option constructor), and it will create the Option instance for you

The other alternative is to pass a list of pre-constructed Option instances to the OptionParser constructor, as in :

```
option_list = [
    make_option("-f", "--filename",
                action="store", type="string", dest="filename"),
    make_option("-q", "--quiet",
                action="store_false", dest="verbose"),
    ]
parser = OptionParser(option_list=option_list)
```

(`make_option()` is a factory function for creating Option instances ; currently it is an alias for the Option constructor. A future version of *optparse* may split Option into several classes, and `make_option()` will pick the right class to instantiate. Do not instantiate Option directly.)

Defining options

Each Option instance represents a set of synonymous command-line option strings, e.g. `-f` and `--file`. You can specify any number of short or long option strings, but you must specify at least one overall option string.

The canonical way to create an *Option* instance is with the `add_option()` method of *OptionParser*.

`OptionParser.`**`add_option`**(*option*)

`OptionParser.`**`add_option`**(**opt_str*, *attr=value*, ...)

To define an option with only a short option string :

```
parser.add_option("-f", attr=value, ...)
```

And to define an option with only a long option string :

```
parser.add_option("--foo", attr=value, ...)
```

The keyword arguments define attributes of the new Option object. The most important option attribute is *action*, and it largely determines which other attributes are relevant or required. If you pass irrelevant option attributes, or fail to pass required ones, *optparse* raises an *OptionError* exception explaining your mistake.

An option's *action* determines what *optparse* does when it encounters this option on the command-line. The standard option actions hard-coded into *optparse* are :

`"store"`
store this option's argument (default)

`"store_const"`
store a constant value, pre-set via *Option.const*

`"store_true"`
store `True`

`"store_false"`
store `False`

`"append"`
append this option's argument to a list

`"append_const"`
append a constant value to a list, pre-set via *Option.const*

`"count"`
increment a counter by one

`"callback"`
call a specified function

`"help"`
print a usage message including all options and the documentation for them

(If you don't supply an action, the default is `"store"`. For this action, you may also supply *type* and *dest* option attributes ; see *Standard option actions*.)

As you can see, most actions involve storing or updating a value somewhere. *optparse* always creates a special object for this, conventionally called `options`, which is an instance of *optparse.Values*.

class optparse.**Values**

An object holding parsed argument names and values as attributes. Normally created by calling when calling *OptionParser.parse_args()*, and can be overridden by a custom subclass passed to the *values* argument of *OptionParser.parse_args()* (as described in *Analyse des arguments*).

Option arguments (and various other values) are stored as attributes of this object, according to the *dest* (destination) option attribute.

For example, when you call

```
parser.parse_args()
```

one of the first things *optparse* does is create the `options` object :

```
options = Values()
```

If one of the options in this parser is defined with

```
parser.add_option("-f", "--file", action="store", type="string", dest="filename")
```

and the command-line being parsed includes any of the following :

```
-ffoo
-f foo
--file=foo
--file foo
```

then *optparse*, on seeing this option, will do the equivalent of

```
options.filename = "foo"
```

The *type* and *dest* option attributes are almost as important as *action*, but *action* is the only one that makes sense for *all* options.

Option attributes

class optparse.**Option**

A single command line argument, with various attributes passed by keyword to the constructor. Normally created with *OptionParser.add_option()* rather than directly, and can be overridden by a custom class via the *option_class* argument to *OptionParser*.

The following option attributes may be passed as keyword arguments to *OptionParser.add_option()*. If you pass an option attribute that is not relevant to a particular option, or fail to pass a required option attribute, *optparse* raises *OptionError*.

Option.**action**

(default : `"store"`)

Determines *optparse*'s behaviour when this option is seen on the command line ; the available options are documented *here*.

Option.**type**

(default : `"string"`)

The argument type expected by this option (e.g., `"string"` or `"int"`) ; the available option types are documented *here*.

`Option.dest`

(default : derived from option strings)

If the option's action implies writing or modifying a value somewhere, this tells *optparse* where to write it : *dest* names an attribute of the `options` object that *optparse* builds as it parses the command line.

`Option.default`

The value to use for this option's destination if the option is not seen on the command line. See also *OptionParser.set_defaults()*.

`Option.nargs`

(default : 1)

How many arguments of type *type* should be consumed when this option is seen. If > 1, *optparse* will store a tuple of values to *dest*.

`Option.const`

For actions that store a constant value, the constant value to store.

`Option.choices`

For options of type `"choice"`, the list of strings the user may choose from.

`Option.callback`

For options with action `"callback"`, the callable to call when this option is seen. See section *Option Callbacks* for detail on the arguments passed to the callable.

`Option.callback_args`

`Option.callback_kwargs`

Additional positional and keyword arguments to pass to `callback` after the four standard callback arguments.

`Option.help`

Help text to print for this option when listing all available options after the user supplies a *help* option (such as `--help`). If no help text is supplied, the option will be listed without help text. To hide this option, use the special value `optparse.SUPPRESS_HELP`.

`Option.metavar`

(default : derived from option strings)

Stand-in for the option argument(s) to use when printing help text. See section *Tutoriel* for an example.

Standard option actions

The various option actions all have slightly different requirements and effects. Most actions have several relevant option attributes which you may specify to guide *optparse*'s behaviour ; a few have required attributes, which you must specify for any option using that action.

— `"store"` [relevant : *type*, *dest*, *nargs*, *choices*]

The option must be followed by an argument, which is converted to a value according to *type* and stored in *dest*. If *nargs* > 1, multiple arguments will be consumed from the command line ; all will be converted according to *type* and stored to *dest* as a tuple. See the *Standard option types* section.

If *choices* is supplied (a list or tuple of strings), the type defaults to `"choice"`.

If *type* is not supplied, it defaults to `"string"`.

If *dest* is not supplied, *optparse* derives a destination from the first long option string (e.g., `--foo-bar` implies `foo_bar`). If there are no long option strings, *optparse* derives a destination from the first short option string (e.g., `-f` implies `f`).

Exemple :

```
parser.add_option("-f")
parser.add_option("-p", type="float", nargs=3, dest="point")
```

As it parses the command line

```
-f foo.txt -p 1 -3.5 4 -fbar.txt
```

optparse will set

```
options.f = "foo.txt"
options.point = (1.0, -3.5, 4.0)
options.f = "bar.txt"
```

— `"store_const"` [required : *const* ; relevant : *dest*]

The value *const* is stored in *dest*.

Exemple :

```
parser.add_option("-q", "--quiet",
                  action="store_const", const=0, dest="verbose")
parser.add_option("-v", "--verbose",
                  action="store_const", const=1, dest="verbose")
parser.add_option("--noisy",
                  action="store_const", const=2, dest="verbose")
```

If `--noisy` is seen, *optparse* will set

```
options.verbose = 2
```

— `"store_true"` [relevant : *dest*]

A special case of `"store_const"` that stores `True` to *dest*.

— `"store_false"` [relevant : *dest*]

Like `"store_true"`, but stores `False`.

Exemple :

```
parser.add_option("--clobber", action="store_true", dest="clobber")
parser.add_option("--no-clobber", action="store_false", dest="clobber")
```

— `"append"` [relevant : *type*, *dest*, *nargs*, *choices*]

The option must be followed by an argument, which is appended to the list in *dest*. If no default value for *dest* is supplied, an empty list is automatically created when *optparse* first encounters this option on the command-line. If *nargs* > 1, multiple arguments are consumed, and a tuple of length *nargs* is appended to *dest*.

The defaults for *type* and *dest* are the same as for the `"store"` action.

Exemple :

```
parser.add_option("-t", "--tracks", action="append", type="int")
```

If `-t3` is seen on the command-line, *optparse* does the equivalent of :

```
options.tracks = []
options.tracks.append(int("3"))
```

If, a little later on, `--tracks=4` is seen, it does :

```
options.tracks.append(int("4"))
```

The `append` action calls the `append` method on the current value of the option. This means that any default value specified must have an `append` method. It also means that if the default value is non-empty, the default elements will be present in the parsed value for the option, with any values from the command line appended after those default values :

```
>>> parser.add_option("--files", action="append", default=['~/.mypkg/defaults
↪'])
>>> opts, args = parser.parse_args(['--files', 'overrides.mypkg'])
>>> opts.files
['~/.mypkg/defaults', 'overrides.mypkg']
```

— `"append_const"` [required : *const* ; relevant : *dest*]

Like `"store_const"`, but the value *const* is appended to *dest* ; as with `"append"`, *dest* defaults to `None`, and an empty list is automatically created the first time the option is encountered.

— `"count"` [relevant : *dest*]

Increment the integer stored at *dest*. If no default value is supplied, *dest* is set to zero before being incremented the first time.

Exemple :

```
parser.add_option("-v", action="count", dest="verbosity")
```

The first time `-v` is seen on the command line, *optparse* does the equivalent of :

```
options.verbosity = 0
options.verbosity += 1
```

Every subsequent occurrence of `-v` results in

```
options.verbosity += 1
```

— `"callback"` [required : *callback* ; relevant : *type*, *nargs*, *callback_args*, *callback_kwargs*]

Call the function specified by *callback*, which is called as

```
func(option, opt_str, value, parser, *args, **kwargs)
```

See section *Option Callbacks* for more detail.

— `"help"`

Prints a complete help message for all the options in the current option parser. The help message is constructed from the `usage` string passed to OptionParser's constructor and the *help* string passed to every option.

If no *help* string is supplied for an option, it will still be listed in the help message. To omit an option entirely, use the special value `optparse.SUPPRESS_HELP`.

optparse automatically adds a *help* option to all OptionParsers, so you do not normally need to create one.

Exemple :

```
from optparse import OptionParser, SUPPRESS_HELP

# usually, a help option is added automatically, but that can
# be suppressed using the add_help_option argument
parser = OptionParser(add_help_option=False)

parser.add_option("-h", "--help", action="help")
parser.add_option("-v", action="store_true", dest="verbose",
                  help="Be moderately verbose")
parser.add_option("--file", dest="filename",
                  help="Input file to read data from")
parser.add_option("--secret", help=SUPPRESS_HELP)
```

If *optparse* sees either `-h` or `--help` on the command line, it will print something like the following help message to stdout (assuming `sys.argv[0]` is `"foo.py"`) :

```
Usage: foo.py [options]

Options:
  -h, --help        Show this help message and exit
  -v                Be moderately verbose
  --file=FILENAME   Input file to read data from
```

After printing the help message, *optparse* terminates your process with `sys.exit(0)`.

— `"version"`

Prints the version number supplied to the OptionParser to stdout and exits. The version number is actually formatted and printed by the `print_version()` method of OptionParser. Generally only relevant if the `version` argument is supplied to the OptionParser constructor. As with *help* options, you will rarely create `version` options, since *optparse* automatically adds them when needed.

Standard option types

optparse has five built-in option types : `"string"`, `"int"`, `"choice"`, `"float"` and `"complex"`. If you need to add new option types, see section *Extending optparse*.

Arguments to string options are not checked or converted in any way : the text on the command line is stored in the destination (or passed to the callback) as-is.

Integer arguments (type `"int"`) are parsed as follows :

- if the number starts with `0x`, it is parsed as a hexadecimal number
- if the number starts with `0`, it is parsed as an octal number
- if the number starts with `0b`, it is parsed as a binary number
- otherwise, the number is parsed as a decimal number

The conversion is done by calling *int()* with the appropriate base (2, 8, 10, or 16). If this fails, so will *optparse*, although with a more useful error message.

`"float"` and `"complex"` option arguments are converted directly with *float()* and *complex()*, with similar error-handling.

`"choice"` options are a subtype of `"string"` options. The *choices* option attribute (a sequence of strings) defines the set of allowed option arguments. `optparse.check_choice()` compares user-supplied option arguments against this master list and raises *OptionValueError* if an invalid string is given.

Analyse des arguments

The whole point of creating and populating an OptionParser is to call its *parse_args()* method.

`OptionParser.`**`parse_args`**(*args=None*, *values=None*)

Parse the command-line options found in *args*.

The input parameters are

`args`
the list of arguments to process (default : `sys.argv[1:]`)

`values`
a *Values* object to store option arguments in (default : a new instance of *Values*) -- if you give an existing object, the option defaults will not be initialized on it

and the return value is a pair `(options, args)` where

`options`
the same object that was passed in as *values*, or the `optparse.Values` instance created by *optparse*

`args`
the leftover positional arguments after all options have been processed

The most common usage is to supply neither keyword argument. If you supply `values`, it will be modified with repeated *setattr()* calls (roughly one for every option argument stored to an option destination) and returned by *parse_args()*.

If *parse_args()* encounters any errors in the argument list, it calls the OptionParser's `error()` method with an appropriate end-user error message. This ultimately terminates your process with an exit status of 2 (the traditional Unix exit status for command-line errors).

Querying and manipulating your option parser

The default behavior of the option parser can be customized slightly, and you can also poke around your option parser and see what's there. OptionParser provides several methods to help you out :

`OptionParser.`**`disable_interspersed_args`**`()`

Set parsing to stop on the first non-option. For example, if `-a` and `-b` are both simple options that take no arguments, *optparse* normally accepts this syntax :

```
prog -a arg1 -b arg2
```

and treats it as equivalent to

```
prog -a -b arg1 arg2
```

To disable this feature, call *disable_interspersed_args()*. This restores traditional Unix syntax, where option parsing stops with the first non-option argument.

Use this if you have a command processor which runs another command which has options of its own and you want to make sure these options don't get confused. For example, each command might have a different set of options.

`OptionParser.`**`enable_interspersed_args`**`()`

Set parsing to not stop on the first non-option, allowing interspersing switches with command arguments. This is the default behavior.

`OptionParser.`**`get_option`**`(`*opt_str*`)`

Returns the Option instance with the option string *opt_str*, or `None` if no options have that option string.

`OptionParser.`**`has_option`**`(`*opt_str*`)`

Return `True` if the OptionParser has an option with option string *opt_str* (e.g., `-q` or `--verbose`).

`OptionParser.`**`remove_option`**`(`*opt_str*`)`

If the *OptionParser* has an option corresponding to *opt_str*, that option is removed. If that option provided any other option strings, all of those option strings become invalid. If *opt_str* does not occur in any option belonging to this *OptionParser*, raises *ValueError*.

Conflicts between options

If you're not careful, it's easy to define options with conflicting option strings :

```
parser.add_option("-n", "--dry-run", ...)
...
parser.add_option("-n", "--noisy", ...)
```

(This is particularly true if you've defined your own OptionParser subclass with some standard options.)

Every time you add an option, *optparse* checks for conflicts with existing options. If it finds any, it invokes the current conflict-handling mechanism. You can set the conflict-handling mechanism either in the constructor :

```
parser = OptionParser(..., conflict_handler=handler)
```

or with a separate call :

```
parser.set_conflict_handler(handler)
```

The available conflict handlers are :

"error" (default)
assume option conflicts are a programming error and raise *OptionConflictError*

"resolve"
resolve option conflicts intelligently (see below)

As an example, let's define an *OptionParser* that resolves conflicts intelligently and add conflicting options to it :

```
parser = OptionParser(conflict_handler="resolve")
parser.add_option("-n", "--dry-run", ..., help="do no harm")
parser.add_option("-n", "--noisy", ..., help="be noisy")
```

At this point, *optparse* detects that a previously added option is already using the `-n` option string. Since `conflict_handler` is `"resolve"`, it resolves the situation by removing `-n` from the earlier option's list of option strings. Now `--dry-run` is the only way for the user to activate that option. If the user asks for help, the help message will reflect that :

```
Options:
  --dry-run     do no harm
  ...
  -n, --noisy   be noisy
```

It's possible to whittle away the option strings for a previously added option until there are none left, and the user has no way of invoking that option from the command-line. In that case, *optparse* removes that option completely, so it doesn't show up in help text or anywhere else. Carrying on with our existing OptionParser :

```
parser.add_option("--dry-run", ..., help="new dry-run option")
```

At this point, the original `-n`/`--dry-run` option is no longer accessible, so *optparse* removes it, leaving this help text :

```
Options:
  ...
  -n, --noisy   be noisy
  --dry-run     new dry-run option
```

Nettoyage

OptionParser instances have several cyclic references. This should not be a problem for Python's garbage collector, but you may wish to break the cyclic references explicitly by calling `destroy()` on your OptionParser once you are done with it. This is particularly useful in long-running applications where large object graphs are reachable from your OptionParser.

Other methods

OptionParser supports several other public methods :

`OptionParser.`**`set_usage`**`(`*usage*`)`

Set the usage string according to the rules described above for the `usage` constructor keyword argument. Passing `None` sets the default usage string ; use `optparse.SUPPRESS_USAGE` to suppress a usage message.

`OptionParser.`**`print_usage`**`(`*file=None*`)`

Print the usage message for the current program (`self.usage`) to *file* (default stdout). Any occurrence of the string `%prog` in `self.usage` is replaced with the name of the current program. Does nothing if `self.usage` is empty or not defined.

`OptionParser.`**`get_usage`**`()`

Same as *print_usage()* but returns the usage string instead of printing it.

OptionParser.**set_defaults**(*dest=value, ...*)

Set default values for several option destinations at once. Using *set_defaults()* is the preferred way to set default values for options, since multiple options can share the same destination. For example, if several "mode" options all set the same destination, any one of them can set the default, and the last one wins :

```
parser.add_option("--advanced", action="store_const",
                  dest="mode", const="advanced",
                  default="novice")     # overridden below
parser.add_option("--novice", action="store_const",
                  dest="mode", const="novice",
                  default="advanced")   # overrides above setting
```

To avoid this confusion, use *set_defaults()* :

```
parser.set_defaults(mode="advanced")
parser.add_option("--advanced", action="store_const",
                  dest="mode", const="advanced")
parser.add_option("--novice", action="store_const",
                  dest="mode", const="novice")
```

17.2.6 Option Callbacks

When *optparse*'s built-in actions and types aren't quite enough for your needs, you have two choices : extend *optparse* or define a callback option. Extending *optparse* is more general, but overkill for a lot of simple cases. Quite often a simple callback is all you need.

There are two steps to defining a callback option :

— define the option itself using the `"callback"` action
— write the callback ; this is a function (or method) that takes at least four arguments, as described below

Defining a callback option

As always, the easiest way to define a callback option is by using the *OptionParser.add_option()* method. Apart from *action*, the only option attribute you must specify is `callback`, the function to call :

```
parser.add_option("-c", action="callback", callback=my_callback)
```

`callback` is a function (or other callable object), so you must have already defined `my_callback()` when you create this callback option. In this simple case, *optparse* doesn't even know if `-c` takes any arguments, which usually means that the option takes no arguments---the mere presence of `-c` on the command-line is all it needs to know. In some circumstances, though, you might want your callback to consume an arbitrary number of command-line arguments. This is where writing callbacks gets tricky ; it's covered later in this section.

optparse always passes four particular arguments to your callback, and it will only pass additional arguments if you specify them via *callback_args* and *callback_kwargs*. Thus, the minimal callback function signature is :

```
def my_callback(option, opt, value, parser):
```

The four arguments to a callback are described below.

There are several other option attributes that you can supply when you define a callback option :

type

has its usual meaning : as with the `"store"` or `"append"` actions, it instructs *optparse* to consume one argument and convert it to *type*. Rather than storing the converted value(s) anywhere, though, *optparse* passes it to your callback function.

nargs

also has its usual meaning : if it is supplied and > 1, *optparse* will consume *nargs* arguments, each of which must be convertible to *type*. It then passes a tuple of converted values to your callback.

callback_args

a tuple of extra positional arguments to pass to the callback

callback_kwargs
a dictionary of extra keyword arguments to pass to the callback

How callbacks are called

All callbacks are called as follows :

```
func(option, opt_str, value, parser, *args, **kwargs)
```

where

option
is the Option instance that's calling the callback

opt_str
is the option string seen on the command-line that's triggering the callback. (If an abbreviated long option was used, `opt_str` will be the full, canonical option string---e.g. if the user puts `--foo` on the command-line as an abbreviation for `--foobar`, then `opt_str` will be `"--foobar"`.)

value
is the argument to this option seen on the command-line. *optparse* will only expect an argument if *type* is set; the type of `value` will be the type implied by the option's type. If *type* for this option is `None` (no argument expected), then `value` will be `None`. If *nargs* > 1, `value` will be a tuple of values of the appropriate type.

parser
is the OptionParser instance driving the whole thing, mainly useful because you can access some other interesting data through its instance attributes :

parser.largs
the current list of leftover arguments, ie. arguments that have been consumed but are neither options nor option arguments. Feel free to modify `parser.largs`, e.g. by adding more arguments to it. (This list will become `args`, the second return value of *parse_args()*.)

parser.rargs
the current list of remaining arguments, ie. with `opt_str` and `value` (if applicable) removed, and only the arguments following them still there. Feel free to modify `parser.rargs`, e.g. by consuming more arguments.

parser.values
the object where option values are by default stored (an instance of optparse.OptionValues). This lets callbacks use the same mechanism as the rest of *optparse* for storing option values; you don't need to mess around with globals or closures. You can also access or modify the value(s) of any options already encountered on the command-line.

args
is a tuple of arbitrary positional arguments supplied via the *callback_args* option attribute.

kwargs
is a dictionary of arbitrary keyword arguments supplied via *callback_kwargs*.

Raising errors in a callback

The callback function should raise *OptionValueError* if there are any problems with the option or its argument(s). *optparse* catches this and terminates the program, printing the error message you supply to stderr. Your message should be clear, concise, accurate, and mention the option at fault. Otherwise, the user will have a hard time figuring out what they did wrong.

Callback example 1 : trivial callback

Here's an example of a callback option that takes no arguments, and simply records that the option was seen :

```
def record_foo_seen(option, opt_str, value, parser):
    parser.values.saw_foo = True

parser.add_option("--foo", action="callback", callback=record_foo_seen)
```

Of course, you could do that with the `"store_true"` action.

Callback example 2 : check option order

Here's a slightly more interesting example : record the fact that `-a` is seen, but blow up if it comes after `-b` in the command-line.

```
def check_order(option, opt_str, value, parser):
    if parser.values.b:
        raise OptionValueError("can't use -a after -b")
    parser.values.a = 1
...
parser.add_option("-a", action="callback", callback=check_order)
parser.add_option("-b", action="store_true", dest="b")
```

Callback example 3 : check option order (generalized)

If you want to reuse this callback for several similar options (set a flag, but blow up if `-b` has already been seen), it needs a bit of work : the error message and the flag that it sets must be generalized.

```
def check_order(option, opt_str, value, parser):
    if parser.values.b:
        raise OptionValueError("can't use %s after -b" % opt_str)
    setattr(parser.values, option.dest, 1)
...
parser.add_option("-a", action="callback", callback=check_order, dest='a')
parser.add_option("-b", action="store_true", dest="b")
parser.add_option("-c", action="callback", callback=check_order, dest='c')
```

Callback example 4 : check arbitrary condition

Of course, you could put any condition in there---you're not limited to checking the values of already-defined options. For example, if you have options that should not be called when the moon is full, all you have to do is this :

```
def check_moon(option, opt_str, value, parser):
    if is_moon_full():
        raise OptionValueError("%s option invalid when moon is full"
                               % opt_str)
    setattr(parser.values, option.dest, 1)
...
parser.add_option("--foo",
                  action="callback", callback=check_moon, dest="foo")
```

(The definition of `is_moon_full()` is left as an exercise for the reader.)

Callback example 5 : fixed arguments

Things get slightly more interesting when you define callback options that take a fixed number of arguments. Specifying that a callback option takes arguments is similar to defining a `"store"` or `"append"` option : if you define *type*, then the option takes one argument that must be convertible to that type ; if you further define *nargs*, then the option takes *nargs* arguments.

Here's an example that just emulates the standard `"store"` action :

```
def store_value(option, opt_str, value, parser):
    setattr(parser.values, option.dest, value)
...
parser.add_option("--foo",
                  action="callback", callback=store_value,
                  type="int", nargs=3, dest="foo")
```

Note that *optparse* takes care of consuming 3 arguments and converting them to integers for you ; all you have to do is store them. (Or whatever ; obviously you don't need a callback for this example.)

Callback example 6 : variable arguments

Things get hairy when you want an option to take a variable number of arguments. For this case, you must write a callback, as *optparse* doesn't provide any built-in capabilities for it. And you have to deal with certain intricacies of conventional Unix command-line parsing that *optparse* normally handles for you. In particular, callbacks should implement the conventional rules for bare `--` and `-` arguments :

— either `--` or `-` can be option arguments
— bare `--` (if not the argument to some option) : halt command-line processing and discard the `--`
— bare `-` (if not the argument to some option) : halt command-line processing but keep the `-` (append it to `parser.largs`)

If you want an option that takes a variable number of arguments, there are several subtle, tricky issues to worry about. The exact implementation you choose will be based on which trade-offs you're willing to make for your application (which is why *optparse* doesn't support this sort of thing directly).

Nevertheless, here's a stab at a callback for an option with variable arguments :

```
def vararg_callback(option, opt_str, value, parser):
    assert value is None
    value = []

    def floatable(str):
        try:
            float(str)
            return True
        except ValueError:
            return False

    for arg in parser.rargs:
        # stop on --foo like options
        if arg[:2] == "--" and len(arg) > 2:
            break
        # stop on -a, but not on -3 or -3.0
        if arg[:1] == "-" and len(arg) > 1 and not floatable(arg):
            break
        value.append(arg)

    del parser.rargs[:len(value)]
    setattr(parser.values, option.dest, value)

...
parser.add_option("-c", "--callback", dest="vararg_attr",
                  action="callback", callback=vararg_callback)
```

17.2.7 Extending `optparse`

Since the two major controlling factors in how *optparse* interprets command-line options are the action and type of each option, the most likely direction of extension is to add new actions and new types.

Adding new types

To add new types, you need to define your own subclass of *optparse*'s *Option* class. This class has a couple of attributes that define *optparse*'s types : *TYPES* and *TYPE_CHECKER*.

`Option.`**`TYPES`**

A tuple of type names ; in your subclass, simply define a new tuple *TYPES* that builds on the standard one.

`Option.TYPE_CHECKER`

A dictionary mapping type names to type-checking functions. A type-checking function has the following signature :

```
def check_mytype(option, opt, value)
```

where `option` is an *Option* instance, `opt` is an option string (e.g., `-f`), and `value` is the string from the command line that must be checked and converted to your desired type. `check_mytype()` should return an object of the hypothetical type `mytype`. The value returned by a type-checking function will wind up in the OptionValues instance returned by *OptionParser.parse_args()*, or be passed to a callback as the `value` parameter.

Your type-checking function should raise *OptionValueError* if it encounters any problems. *OptionValueError* takes a single string argument, which is passed as-is to *OptionParser*'s `error()` method, which in turn prepends the program name and the string `"error:"` and prints everything to stderr before terminating the process.

Here's a silly example that demonstrates adding a `"complex"` option type to parse Python-style complex numbers on the command line. (This is even sillier than it used to be, because *optparse* 1.3 added built-in support for complex numbers, but never mind.)

First, the necessary imports :

```
from copy import copy
from optparse import Option, OptionValueError
```

You need to define your type-checker first, since it's referred to later (in the *TYPE_CHECKER* class attribute of your Option subclass) :

```
def check_complex(option, opt, value):
    try:
        return complex(value)
    except ValueError:
        raise OptionValueError(
            "option %s: invalid complex value: %r" % (opt, value))
```

Finally, the Option subclass :

```
class MyOption (Option):
    TYPES = Option.TYPES + ("complex",)
    TYPE_CHECKER = copy(Option.TYPE_CHECKER)
    TYPE_CHECKER["complex"] = check_complex
```

(If we didn't make a *copy()* of *Option.TYPE_CHECKER*, we would end up modifying the *TYPE_CHECKER* attribute of *optparse*'s Option class. This being Python, nothing stops you from doing that except good manners and common sense.)

That's it ! Now you can write a script that uses the new option type just like any other *optparse*-based script, except you have to instruct your OptionParser to use MyOption instead of Option :

```
parser = OptionParser(option_class=MyOption)
parser.add_option("-c", type="complex")
```

Alternately, you can build your own option list and pass it to OptionParser ; if you don't use `add_option()` in the above way, you don't need to tell OptionParser which option class to use :

```
option_list = [MyOption("-c", action="store", type="complex", dest="c")]
parser = OptionParser(option_list=option_list)
```

Adding new actions

Adding new actions is a bit trickier, because you have to understand that *optparse* has a couple of classifications for actions :

"store" actions
actions that result in *optparse* storing a value to an attribute of the current OptionValues instance ; these options require a *dest* attribute to be supplied to the Option constructor.

"typed" actions
actions that take a value from the command line and expect it to be of a certain type ; or rather, a string that can be converted to a certain type. These options require a *type* attribute to the Option constructor.

These are overlapping sets : some default "store" actions are `"store"`, `"store_const"`, `"append"`, and `"count"`, while the default "typed" actions are `"store"`, `"append"`, and `"callback"`.

When you add an action, you need to categorize it by listing it in at least one of the following class attributes of Option (all are lists of strings) :

`Option.`**`ACTIONS`**
All actions must be listed in ACTIONS.

`Option.`**`STORE_ACTIONS`**
"store" actions are additionally listed here.

`Option.`**`TYPED_ACTIONS`**
"typed" actions are additionally listed here.

`Option.`**`ALWAYS_TYPED_ACTIONS`**
Actions that always take a type (i.e. whose options always take a value) are additionally listed here. The only effect of this is that *optparse* assigns the default type, `"string"`, to options with no explicit type whose action is listed in *ALWAYS_TYPED_ACTIONS*.

In order to actually implement your new action, you must override Option's `take_action()` method and add a case that recognizes your action.

For example, let's add an `"extend"` action. This is similar to the standard `"append"` action, but instead of taking a single value from the command-line and appending it to an existing list, `"extend"` will take multiple values in a single comma-delimited string, and extend an existing list with them. That is, if `--names` is an `"extend"` option of type `"string"`, the command line

```
--names=foo,bar --names blah --names ding,dong
```

would result in a list

```
["foo", "bar", "blah", "ding", "dong"]
```

Again we define a subclass of Option :

```
class MyOption(Option):

    ACTIONS = Option.ACTIONS + ("extend",)
    STORE_ACTIONS = Option.STORE_ACTIONS + ("extend",)
    TYPED_ACTIONS = Option.TYPED_ACTIONS + ("extend",)
    ALWAYS_TYPED_ACTIONS = Option.ALWAYS_TYPED_ACTIONS + ("extend",)

    def take_action(self, action, dest, opt, value, values, parser):
        if action == "extend":
            lvalue = value.split(",")
            values.ensure_value(dest, []).extend(lvalue)
        else:
            Option.take_action(
                self, action, dest, opt, value, values, parser)
```

Features of note :

- "extend" both expects a value on the command-line and stores that value somewhere, so it goes in both *STORE_ACTIONS* and *TYPED_ACTIONS*.
- to ensure that *optparse* assigns the default type of "string" to "extend" actions, we put the "extend" action in *ALWAYS_TYPED_ACTIONS* as well.
- MyOption.take_action() implements just this one new action, and passes control back to Option.take_action() for the standard *optparse* actions.
- values is an instance of the optparse_parser.Values class, which provides the very useful ensure_value() method. ensure_value() is essentially *getattr()* with a safety valve ; it is called as

```
values.ensure_value(attr, value)
```

If the attr attribute of values doesn't exist or is None, then ensure_value() first sets it to value, and then returns value. This is very handy for actions like "extend", "append", and "count", all of which accumulate data in a variable and expect that variable to be of a certain type (a list for the first two, an integer for the latter). Using ensure_value() means that scripts using your action don't have to worry about setting a default value for the option destinations in question ; they can just leave the default as None and ensure_value() will take care of getting it right when it's needed.

17.2.8 Exceptions

exception optparse.**OptionError**

Raised if an *Option* instance is created with invalid or inconsistent arguments.

exception optparse.**OptionConflictError**

Raised if conflicting options are added to an *OptionParser*.

exception optparse.**OptionValueError**

Raised if an invalid option value is encountered on the command line.

exception optparse.**BadOptionError**

Raised if an invalid option is passed on the command line.

exception optparse.**AmbiguousOptionError**

Raised if an ambiguous option is passed on the command line.

17.3 getpass --- Portable password input

Source code : Lib/getpass.py

Availability : not WASI.

This module does not work or is not available on WebAssembly. See *Plateformes WebAssembly* for more information.

Le module *getpass* fournit 2 fonctions :

getpass.**getpass**(*prompt='Password : '*, *stream=None*)

Affiche une demande de mot de passe sans renvoyer d'écho. L'utilisateur est invité en utilisant la string *prompt*, avec en valeur par défaut 'Password: '. Avec Unix, l'invite est écrite dans l'objet fichier *stream* en utilisant si besoin le *replace error handler*. *stream* sera par défaut le terminal de contrôle (/dev/tty), ou si celui ci n'est pas disponible ce sera sys.stderr (cet argument sera ignoré sur Windows).

Si aucune saisie en mode sans affichage n'est disponible, getpass() se résoudra à afficher un message d'avertissement vers *stream*, puis lire l'entrée depuis sys.stdin, en levant une *GetPassWarning*.

> **Note**
>
> Si vous appelez *getpass* depuis IDLE, la saisie peut être faite dans le terminal depuis lequel IDLE a été lancé, plutôt que dans la fenêtre d'IDLE.

exception getpass.**GetPassWarning**

Une sous classe d'exception `UserWarning` est levée quand le mot de passe saisi pourrait être affiché.

getpass.**getuser**()

Renvoie le *login name* de l'utilisateur.

This function checks the environment variables `LOGNAME`, `USER`, `LNAME` and `USERNAME`, in order, and returns the value of the first one which is set to a non-empty string. If none are set, the login name from the password database is returned on systems which support the `pwd` module, otherwise, an `OSError` is raised.

In general, this function should be preferred over `os.getlogin()`.

Modifié dans la version 3.13 : Previously, various exceptions beyond just `OSError` were raised.

17.4 `fileinput` --- Iterate over lines from multiple input streams

Code source : Lib/fileinput.py

Ce module offre une classe auxiliaire et des fonctions pour lire facilement l'entrée standard ou bien les fichiers d'une liste. Si vous n'avez besoin de lire ou écrire qu'un seul fichier, il suffit de `open()`.

Ce module s'utilise le plus couramment comme ceci :

```
import fileinput
for line in fileinput.input(encoding="utf-8"):
    process(line)
```

Ce code traite une à une les lignes des fichiers de `sys.argv[1:]`. Si cette liste est vide (pas d'argument en ligne de commande), il lit l'entrée standard. Le nom de fichier `'-'` est équivalent à l'entrée standard (les arguments facultatifs *mode* et *openhook* sont ignorés dans ce cas). On peut aussi passer la liste des fichiers comme argument à `input()`, voire un nom de fichier unique.

Par défaut, tous les fichiers sont ouverts en mode texte. On peut changer ce comportement à l'aide du paramètre *mode* de la fonction `input()` ou du constructeur de `FileInput`. Si une erreur d'entrée-sortie se produit durant l'ouverture ou la lecture d'un fichier, l'exception `OSError` est levée.

Modifié dans la version 3.3 : `IOError` était levée auparavant, elle est devenue un alias de `OSError`.

Si `sys.stdin` apparaît plus d'une fois dans la liste, toutes les lignes sont consommées dès la première fois, sauf éventuellement en cas d'usage interactif ou si le flux d'entrée standard a été modifié dans l'intervalle (par exemple avec `sys.stdin.seek(0)`).

Les fichiers vides sont ouverts et refermés immédiatement. Ils ne sont pas détectables dans la liste des fichiers, sauf éventuellement dans le cas où le dernier fichier est vide.

Les caractères de saut de ligne sont préservés, donc toutes les lignes se terminent par un saut de ligne, sauf éventuellement la dernière ligne d'un fichier.

You can control how files are opened by providing an opening hook via the *openhook* parameter to `fileinput.input()` or `FileInput()`. The hook must be a function that takes two arguments, *filename* and *mode*, and returns an accordingly opened file-like object. If *encoding* and/or *errors* are specified, they will be passed to the hook as additional keyword arguments. This module provides a `hook_compressed()` to support compressed files.

La fonction suivante constitue l'interface principale du module :

fileinput.**input**(*files=None*, *inplace=False*, *backup=''*, ***, *mode='r'*, *openhook=None*, *encoding=None*, *errors=None*)

Crée une instance de `FileInput`, qui devient l'état global pour toutes les fonctions du module. Elle est également renvoyée afin que l'utilisateur puisse la parcourir comme un objet itérable. Tous les paramètres de cette fonction sont transmis au constructeur de `FileInput`.

Les instances de `FileInput` peuvent s'utiliser comme gestionnaires de contexte, avec l'instruction `with`. Dans le code suivant, *input* est fermé lorsque le bloc `with` se termine, y compris si une exception l'a interrompu :

```
with fileinput.input(files=('spam.txt', 'eggs.txt'), encoding="utf-8") as f:
    for line in f:
        process(line)
```

Modifié dans la version 3.2 : prise en charge du protocole de gestionnaire de contexte.

Modifié dans la version 3.8 : les paramètres *mode* et *openhook* doivent être nommés.

Modifié dans la version 3.10 : The keyword-only parameter *encoding* and *errors* are added.

Toutes les fonctions suivantes font appel à l'état global du module mis en place par `fileinput.input()`. L'absence de cet état déclenche l'exception `RuntimeError`.

`fileinput.filename()`

Renvoie le nom du fichier en train d'être lu, ou `None` avant la lecture de la toute première ligne.

`fileinput.fileno()`

Renvoie le descripteur de fichier (sous forme d'entier) utilisé pour lire le fichier courant. Si aucun fichier n'est ouvert (avant la première ligne et entre les fichiers), le résultat est `-1`.

`fileinput.lineno()`

Renvoie le numéro de la ligne qui vient d'être lue, en commençant par la première ligne du premier fichier. Avant cette toute première ligne, renvoie `0`. Après la dernière ligne du dernier fichier, renvoie le numéro de cette ligne.

`fileinput.filelineno()`

Renvoie le numéro de ligne relatif au fichier courant. Avant la toute première ligne, renvoie `0`. Après la toute dernière ligne, renvoie le numéro de cette ligne par rapport à son fichier source.

`fileinput.isfirstline()`

Renvoie `True` ou `False` selon que la ligne qui vient d'être lue est la première du fichier.

`fileinput.isstdin()`

`True` ou `False` selon que la dernière ligne lue provenait de `sys.stdin` ou non.

`fileinput.nextfile()`

Ferme le fichier courant et laisse la lecture se poursuivre au début du suivant (ou se terminer si c'était le dernier fichier ; dans ce cas cette fonction ne fait rien). Les lignes court-circuitées ne comptent pas dans les numéros des lignes des fichiers suivants. Le nom du fichier courant n'est pas modifié immédiatement, mais seulement après que la première ligne du fichier suivant a été lue. Cette fonction n'a pas d'effet avant la lecture de la première ligne (elle ne peut pas sauter le premier fichier).

`fileinput.close()`

Ferme le fichier courant et termine la lecture en sautant les fichiers suivants.

La classe qui implémente ce comportement du module est publique. On peut en créer des classes filles :

class `fileinput.FileInput`(*files=None*, *inplace=False*, *backup=''*, ***, *mode='r'*, *openhook=None*, *encoding=None*, *errors=None*)

Class `FileInput` is the implementation ; its methods `filename()`, `fileno()`, `lineno()`, `filelineno()`, `isfirstline()`, `isstdin()`, `nextfile()` and `close()` correspond to the functions of the same name in the module. In addition it is *iterable* and has a `readline()` method which returns the next input line. The sequence must be accessed in strictly sequential order ; random access and `readline()` cannot be mixed.

With *mode* you can specify which file mode will be passed to `open()`. It must be one of `'r'` and `'rb'`.

S'il est fourni, l'argument *openhook* est une fonction. Elle est appelée avec les paramètres *filename* et *mode*, et renvoie un objet fichier-compatible ouvert selon *mode*. Notez que *openhook* et *inplace* sont mutuellement exclusifs.

You can specify *encoding* and *errors* that is passed to `open()` or *openhook*.

Les objets `FileInput` peuvent aussi fonctionner comme gestionnaires de contexte dans un bloc `with`. Dans l'exemple suivant, *input* est fermé à la fin du bloc `with`, même arrêté par une exception :

```
with FileInput(files=('spam.txt', 'eggs.txt')) as input:
    process(input)
```

Modifié dans la version 3.2 : prise en charge du protocole de gestionnaire de contexte.

Modifié dans la version 3.8 : les paramètres *mode* et *openhook* doivent impérativement être nommés.

Modifié dans la version 3.10 : The keyword-only parameter *encoding* and *errors* are added.

Modifié dans la version 3.11 : The `'rU'` and `'U'` modes and the `__getitem__()` method have been removed.

Filtrage sur place optionnel : si `inplace=True` est passé à `fileinput.input()` ou au constructeur de `FileInput`, chaque fichier d'entrée est déplacé vers une sauvegarde et la sortie standard est redirigée vers le fichier lui-même, ce qui permet d'écrire des filtres qui modifient directement les fichiers existants. Si le paramètre *backup* est fourni, il donne l'extension des fichiers de sauvegarde. Dans ce cas, la sauvegarde est conservée après l'opération. Par défaut, les fichiers de sauvegarde ont l'extension `'.bak'` et sont supprimés dès que le fichier de sortie est fermé. Si un fichier existe avec le même nom que la sauvegarde, il est écrasé. Le filtrage sur place ne fait rien pour l'entrée standard.

Les deux fonctions suivantes fournissent des valeurs prédéfinies pour *openhook* :

`fileinput.`**`hook_compressed`**(*filename*, *mode*, ***, *encoding=None*, *errors=None*)

Ouvre de façon transparente les fichiers compressés avec gzip ou bzip2, à l'aide des modules `gzip` et `bz2`. Les fichiers compressés sont reconnus aux extensions `'.gz'` et `'.bz2'`. Tous les fichiers qui n'ont pas l'une de ces deux extensions sont ouverts normalement (avec `open()`, sans décompression).

The *encoding* and *errors* values are passed to `io.TextIOWrapper` for compressed files and open for normal files.

Usage example : `fi = fileinput.FileInput(openhook=fileinput.hook_compressed, encoding="utf-8")`

Modifié dans la version 3.10 : The keyword-only parameter *encoding* and *errors* are added.

`fileinput.`**`hook_encoded`**(*encoding*, *errors=None*)

Renvoie une fonction qui ouvre les fichiers en passant à `open()` les arguments *encoding* et *errors*. Le résultat peut être exploité à travers le point d'entrée automatique *openhook*.

Exemple d'utilisation : `fi = fileinput.FileInput(openhook=fileinput.hook_encoded("utf-8", "surrogateescape"))`.

Modifié dans la version 3.6 : ajout du paramètre facultatif *errors*.

Obsolète depuis la version 3.10 : This function is deprecated since `fileinput.input()` and `FileInput` now have *encoding* and *errors* parameters.

17.5 `curses` --- Terminal handling for character-cell displays

Source code : Lib/curses

The `curses` module provides an interface to the curses library, the de-facto standard for portable advanced terminal handling.

While curses is most widely used in the Unix environment, versions are available for Windows, DOS, and possibly other systems as well. This extension module is designed to match the API of ncurses, an open-source curses library hosted on Linux and the BSD variants of Unix.

Availability : not Android, not iOS, not WASI.

This module is not supported on *mobile platforms* or *WebAssembly platforms*.

Note

Whenever the documentation mentions a *character* it can be specified as an integer, a one-character Unicode string or a one-byte byte string.

Whenever the documentation mentions a *character string* it can be specified as a Unicode string or a byte string.

Voir aussi

Module `curses.ascii`
Utilities for working with ASCII characters, regardless of your locale settings.

Module `curses.panel`
A panel stack extension that adds depth to curses windows.

Module `curses.textpad`
Editable text widget for curses supporting **Emacs**-like bindings.

curses-howto
Tutorial material on using curses with Python, by Andrew Kuchling and Eric Raymond.

17.5.1 Fonctions

The module `curses` defines the following exception :

exception `curses.error`

Exception raised when a curses library function returns an error.

Note

Whenever *x* or *y* arguments to a function or a method are optional, they default to the current cursor location. Whenever *attr* is optional, it defaults to `A_NORMAL`.

The module `curses` defines the following functions :

`curses.baudrate()`

Return the output speed of the terminal in bits per second. On software terminal emulators it will have a fixed high value. Included for historical reasons ; in former times, it was used to write output loops for time delays and occasionally to change interfaces depending on the line speed.

`curses.beep()`

Emit a short attention sound.

`curses.can_change_color()`

Return `True` or `False`, depending on whether the programmer can change the colors displayed by the terminal.

`curses.cbreak()`

Enter cbreak mode. In cbreak mode (sometimes called "rare" mode) normal tty line buffering is turned off and characters are available to be read one by one. However, unlike raw mode, special characters (interrupt, quit, suspend, and flow control) retain their effects on the tty driver and calling program. Calling first `raw()` then `cbreak()` leaves the terminal in cbreak mode.

`curses.color_content(color_number)`

Return the intensity of the red, green, and blue (RGB) components in the color *color_number*, which must be between `0` and `COLORS - 1`. Return a 3-tuple, containing the R,G,B values for the given color, which will be between `0` (no component) and `1000` (maximum amount of component).

curses.**color_pair**(*pair_number*)

Return the attribute value for displaying text in the specified color pair. Only the first 256 color pairs are supported. This attribute value can be combined with *A_STANDOUT*, *A_REVERSE*, and the other A_* attributes. *pair_number()* is the counterpart to this function.

curses.**curs_set**(*visibility*)

Set the cursor state. *visibility* can be set to 0, 1, or 2, for invisible, normal, or very visible. If the terminal supports the visibility requested, return the previous cursor state ; otherwise raise an exception. On many terminals, the "visible" mode is an underline cursor and the "very visible" mode is a block cursor.

curses.**def_prog_mode**()

Save the current terminal mode as the "program" mode, the mode when the running program is using curses. (Its counterpart is the "shell" mode, for when the program is not in curses.) Subsequent calls to *reset_prog_mode()* will restore this mode.

curses.**def_shell_mode**()

Save the current terminal mode as the "shell" mode, the mode when the running program is not using curses. (Its counterpart is the "program" mode, when the program is using curses capabilities.) Subsequent calls to *reset_shell_mode()* will restore this mode.

curses.**delay_output**(*ms*)

Insert an *ms* millisecond pause in output.

curses.**doupdate**()

Update the physical screen. The curses library keeps two data structures, one representing the current physical screen contents and a virtual screen representing the desired next state. The *doupdate()* ground updates the physical screen to match the virtual screen.

The virtual screen may be updated by a *noutrefresh()* call after write operations such as *addstr()* have been performed on a window. The normal *refresh()* call is simply noutrefresh() followed by doupdate() ; if you have to update multiple windows, you can speed performance and perhaps reduce screen flicker by issuing noutrefresh() calls on all windows, followed by a single doupdate().

curses.**echo**()

Enter echo mode. In echo mode, each character input is echoed to the screen as it is entered.

curses.**endwin**()

De-initialize the library, and return terminal to normal status.

curses.**erasechar**()

Return the user's current erase character as a one-byte bytes object. Under Unix operating systems this is a property of the controlling tty of the curses program, and is not set by the curses library itself.

curses.**filter**()

The *filter()* routine, if used, must be called before *initscr()* is called. The effect is that, during those calls, LINES is set to 1 ; the capabilities clear, cup, cud, cud1, cuu1, cuu, vpa are disabled ; and the home string is set to the value of cr. The effect is that the cursor is confined to the current line, and so are screen updates. This may be used for enabling character-at-a-time line editing without touching the rest of the screen.

curses.**flash**()

Flash the screen. That is, change it to reverse-video and then change it back in a short interval. Some people prefer such as 'visible bell' to the audible attention signal produced by *beep()*.

curses.**flushinp**()

Flush all input buffers. This throws away any typeahead that has been typed by the user and has not yet been processed by the program.

curses.**getmouse**()

After *getch()* returns *KEY_MOUSE* to signal a mouse event, this method should be called to retrieve the queued mouse event, represented as a 5-tuple (id, x, y, z, bstate). *id* is an ID value used to distinguish multiple devices, and *x*, *y*, *z* are the event's coordinates. (*z* is currently unused.) *bstate* is an integer value whose bits will be set to indicate the type of event, and will be the bitwise OR of one

or more of the following constants, where *n* is the button number from 1 to 5 : *BUTTONn_PRESSED*, *BUTTONn_RELEASED*, *BUTTONn_CLICKED*, *BUTTONn_DOUBLE_CLICKED*, *BUTTONn_TRIPLE_CLICKED*, *BUTTON_SHIFT*, *BUTTON_CTRL*, *BUTTON_ALT*.

Modifié dans la version 3.10 : The `BUTTON5_*` constants are now exposed if they are provided by the underlying curses library.

`curses.`**`getsyx`**`()`

Return the current coordinates of the virtual screen cursor as a tuple `(y, x)`. If *leaveok* is currently `True`, then return `(-1, -1)`.

`curses.`**`getwin`**`(`*file*`)`

Read window related data stored in the file by an earlier `window.putwin()` call. The routine then creates and initializes a new window using that data, returning the new window object.

`curses.`**`has_colors`**`()`

Return `True` if the terminal can display colors ; otherwise, return `False`.

`curses.`**`has_extended_color_support`**`()`

Return `True` if the module supports extended colors ; otherwise, return `False`. Extended color support allows more than 256 color pairs for terminals that support more than 16 colors (e.g. xterm-256color).

Extended color support requires ncurses version 6.1 or later.

Ajouté dans la version 3.10.

`curses.`**`has_ic`**`()`

Return `True` if the terminal has insert- and delete-character capabilities. This function is included for historical reasons only, as all modern software terminal emulators have such capabilities.

`curses.`**`has_il`**`()`

Return `True` if the terminal has insert- and delete-line capabilities, or can simulate them using scrolling regions. This function is included for historical reasons only, as all modern software terminal emulators have such capabilities.

`curses.`**`has_key`**`(`*ch*`)`

Take a key value *ch*, and return `True` if the current terminal type recognizes a key with that value.

`curses.`**`halfdelay`**`(`*tenths*`)`

Used for half-delay mode, which is similar to cbreak mode in that characters typed by the user are immediately available to the program. However, after blocking for *tenths* tenths of seconds, raise an exception if nothing has been typed. The value of *tenths* must be a number between `1` and `255`. Use `nocbreak()` to leave half-delay mode.

`curses.`**`init_color`**`(`*color_number*, *r*, *g*, *b*`)`

Change the definition of a color, taking the number of the color to be changed followed by three RGB values (for the amounts of red, green, and blue components). The value of *color_number* must be between `0` and `COLORS - 1`. Each of *r*, *g*, *b*, must be a value between `0` and `1000`. When `init_color()` is used, all occurrences of that color on the screen immediately change to the new definition. This function is a no-op on most terminals ; it is active only if `can_change_color()` returns `True`.

`curses.`**`init_pair`**`(`*pair_number*, *fg*, *bg*`)`

Change the definition of a color-pair. It takes three arguments : the number of the color-pair to be changed, the foreground color number, and the background color number. The value of *pair_number* must be between `1` and `COLOR_PAIRS - 1` (the `0` color pair is wired to white on black and cannot be changed). The value of *fg* and *bg* arguments must be between `0` and `COLORS - 1`, or, after calling `use_default_colors()`, `-1`. If the color-pair was previously initialized, the screen is refreshed and all occurrences of that color-pair are changed to the new definition.

`curses.`**`initscr`**`()`

Initialize the library. Return a *window* object which represents the whole screen.

> **Note**
>
> If there is an error opening the terminal, the underlying curses library may cause the interpreter to exit.

`curses.is_term_resized(nlines, ncols)`

Return `True` if `resize_term()` would modify the window structure, `False` otherwise.

`curses.isendwin()`

Return `True` if `endwin()` has been called (that is, the curses library has been deinitialized).

`curses.keyname(k)`

Return the name of the key numbered *k* as a bytes object. The name of a key generating printable ASCII character is the key's character. The name of a control-key combination is a two-byte bytes object consisting of a caret (`b'^'`) followed by the corresponding printable ASCII character. The name of an alt-key combination (128--255) is a bytes object consisting of the prefix `b'M-'` followed by the name of the corresponding ASCII character.

`curses.killchar()`

Return the user's current line kill character as a one-byte bytes object. Under Unix operating systems this is a property of the controlling tty of the curses program, and is not set by the curses library itself.

`curses.longname()`

Return a bytes object containing the terminfo long name field describing the current terminal. The maximum length of a verbose description is 128 characters. It is defined only after the call to `initscr()`.

`curses.meta(flag)`

If *flag* is `True`, allow 8-bit characters to be input. If *flag* is `False`, allow only 7-bit chars.

`curses.mouseinterval(interval)`

Set the maximum time in milliseconds that can elapse between press and release events in order for them to be recognized as a click, and return the previous interval value. The default value is 200 milliseconds, or one fifth of a second.

`curses.mousemask(mousemask)`

Set the mouse events to be reported, and return a tuple `(availmask, oldmask)`. *availmask* indicates which of the specified mouse events can be reported; on complete failure it returns `0`. *oldmask* is the previous value of the given window's mouse event mask. If this function is never called, no mouse events are ever reported.

`curses.napms(ms)`

Sleep for *ms* milliseconds.

`curses.newpad(nlines, ncols)`

Create and return a pointer to a new pad data structure with the given number of lines and columns. Return a pad as a window object.

A pad is like a window, except that it is not restricted by the screen size, and is not necessarily associated with a particular part of the screen. Pads can be used when a large window is needed, and only a part of the window will be on the screen at one time. Automatic refreshes of pads (such as from scrolling or echoing of input) do not occur. The `refresh()` and `noutrefresh()` methods of a pad require 6 arguments to specify the part of the pad to be displayed and the location on the screen to be used for the display. The arguments are *pminrow*, *pmincol*, *sminrow*, *smincol*, *smaxrow*, *smaxcol*; the *p* arguments refer to the upper left corner of the pad region to be displayed and the *s* arguments define a clipping box on the screen within which the pad region is to be displayed.

`curses.newwin(nlines, ncols)`

`curses.newwin(nlines, ncols, begin_y, begin_x)`

Return a new *window*, whose left-upper corner is at `(begin_y, begin_x)`, and whose height/width is *nlines*/*ncols*.

By default, the window will extend from the specified position to the lower right corner of the screen.

`curses.nl()`

Enter newline mode. This mode translates the return key into newline on input, and translates newline into return and line-feed on output. Newline mode is initially on.

`curses.nocbreak()`

Leave cbreak mode. Return to normal "cooked" mode with line buffering.

`curses.noecho()`

Leave echo mode. Echoing of input characters is turned off.

`curses.nonl()`

Leave newline mode. Disable translation of return into newline on input, and disable low-level translation of newline into newline/return on output (but this does not change the behavior of `addch('\n')`, which always does the equivalent of return and line feed on the virtual screen). With translation off, curses can sometimes speed up vertical motion a little ; also, it will be able to detect the return key on input.

`curses.noqiflush()`

When the `noqiflush()` routine is used, normal flush of input and output queues associated with the `INTR`, `QUIT` and `SUSP` characters will not be done. You may want to call `noqiflush()` in a signal handler if you want output to continue as though the interrupt had not occurred, after the handler exits.

`curses.noraw()`

Leave raw mode. Return to normal "cooked" mode with line buffering.

`curses.pair_content(pair_number)`

Return a tuple `(fg, bg)` containing the colors for the requested color pair. The value of *pair_number* must be between `0` and `COLOR_PAIRS - 1`.

`curses.pair_number(attr)`

Return the number of the color-pair set by the attribute value *attr*. `color_pair()` is the counterpart to this function.

`curses.putp(str)`

Equivalent to `tputs(str, 1, putchar)` ; emit the value of a specified terminfo capability for the current terminal. Note that the output of `putp()` always goes to standard output.

`curses.qiflush([flag])`

If *flag* is `False`, the effect is the same as calling `noqiflush()`. If *flag* is `True`, or no argument is provided, the queues will be flushed when these control characters are read.

`curses.raw()`

Enter raw mode. In raw mode, normal line buffering and processing of interrupt, quit, suspend, and flow control keys are turned off ; characters are presented to curses input functions one by one.

`curses.reset_prog_mode()`

Restore the terminal to "program" mode, as previously saved by `def_prog_mode()`.

`curses.reset_shell_mode()`

Restore the terminal to "shell" mode, as previously saved by `def_shell_mode()`.

`curses.resetty()`

Restore the state of the terminal modes to what it was at the last call to `savetty()`.

`curses.resize_term(nlines, ncols)`

Backend function used by `resizeterm()`, performing most of the work ; when resizing the windows, `resize_term()` blank-fills the areas that are extended. The calling application should fill in these areas with appropriate data. The `resize_term()` function attempts to resize all windows. However, due to the calling convention of pads, it is not possible to resize these without additional interaction with the application.

`curses.resizeterm(nlines, ncols)`

Resize the standard and current windows to the specified dimensions, and adjusts other bookkeeping data used by the curses library that record the window dimensions (in particular the SIGWINCH handler).

curses.**savetty**()

Save the current state of the terminal modes in a buffer, usable by *resetty()*.

curses.**get_escdelay**()

Retrieves the value set by *set_escdelay()*.

Ajouté dans la version 3.9.

curses.**set_escdelay**(*ms*)

Sets the number of milliseconds to wait after reading an escape character, to distinguish between an individual escape character entered on the keyboard from escape sequences sent by cursor and function keys.

Ajouté dans la version 3.9.

curses.**get_tabsize**()

Retrieves the value set by *set_tabsize()*.

Ajouté dans la version 3.9.

curses.**set_tabsize**(*size*)

Sets the number of columns used by the curses library when converting a tab character to spaces as it adds the tab to a window.

Ajouté dans la version 3.9.

curses.**setsyx**(*y*, *x*)

Set the virtual screen cursor to *y*, *x*. If *y* and *x* are both `-1`, then *leaveok* is set `True`.

curses.**setupterm**(*term=None*, *fd=-1*)

Initialize the terminal. *term* is a string giving the terminal name, or `None`; if omitted or `None`, the value of the `TERM` environment variable will be used. *fd* is the file descriptor to which any initialization sequences will be sent; if not supplied or `-1`, the file descriptor for `sys.stdout` will be used.

curses.**start_color**()

Must be called if the programmer wants to use colors, and before any other color manipulation routine is called. It is good practice to call this routine right after *initscr()*.

start_color() initializes eight basic colors (black, red, green, yellow, blue, magenta, cyan, and white), and two global variables in the *curses* module, *COLORS* and *COLOR_PAIRS*, containing the maximum number of colors and color-pairs the terminal can support. It also restores the colors on the terminal to the values they had when the terminal was just turned on.

curses.**termattrs**()

Return a logical OR of all video attributes supported by the terminal. This information is useful when a curses program needs complete control over the appearance of the screen.

curses.**termname**()

Return the value of the environment variable `TERM`, as a bytes object, truncated to 14 characters.

curses.**tigetflag**(*capname*)

Return the value of the Boolean capability corresponding to the terminfo capability name *capname* as an integer. Return the value `-1` if *capname* is not a Boolean capability, or `0` if it is canceled or absent from the terminal description.

curses.**tigetnum**(*capname*)

Return the value of the numeric capability corresponding to the terminfo capability name *capname* as an integer. Return the value `-2` if *capname* is not a numeric capability, or `-1` if it is canceled or absent from the terminal description.

curses.**tigetstr**(*capname*)

Return the value of the string capability corresponding to the terminfo capability name *capname* as a bytes object. Return `None` if *capname* is not a terminfo "string capability", or is canceled or absent from the terminal description.

`curses.tparm(str[, ...])`

Instantiate the bytes object *str* with the supplied parameters, where *str* should be a parameterized string obtained from the terminfo database. E.g. `tparm(tigetstr("cup"), 5, 3)` could result in `b'\033[6;4H'`, the exact result depending on terminal type.

`curses.typeahead(fd)`

Specify that the file descriptor *fd* be used for typeahead checking. If *fd* is `-1`, then no typeahead checking is done.

The curses library does "line-breakout optimization" by looking for typeahead periodically while updating the screen. If input is found, and it is coming from a tty, the current update is postponed until refresh or doupdate is called again, allowing faster response to commands typed in advance. This function allows specifying a different file descriptor for typeahead checking.

`curses.unctrl(ch)`

Return a bytes object which is a printable representation of the character *ch*. Control characters are represented as a caret followed by the character, for example as `b'^C'`. Printing characters are left as they are.

`curses.ungetch(ch)`

Push *ch* so the next `getch()` will return it.

> **Note**
>
> Only one *ch* can be pushed before `getch()` is called.

`curses.update_lines_cols()`

Update the `LINES` and `COLS` module variables. Useful for detecting manual screen resize.

Ajouté dans la version 3.5.

`curses.unget_wch(ch)`

Push *ch* so the next `get_wch()` will return it.

> **Note**
>
> Only one *ch* can be pushed before `get_wch()` is called.

Ajouté dans la version 3.3.

`curses.ungetmouse(id, x, y, z, bstate)`

Push a `KEY_MOUSE` event onto the input queue, associating the given state data with it.

`curses.use_env(flag)`

If used, this function should be called before `initscr()` or newterm are called. When *flag* is `False`, the values of lines and columns specified in the terminfo database will be used, even if environment variables `LINES` and `COLUMNS` (used by default) are set, or if curses is running in a window (in which case default behavior would be to use the window size if `LINES` and `COLUMNS` are not set).

`curses.use_default_colors()`

Allow use of default values for colors on terminals supporting this feature. Use this to support transparency in your application. The default color is assigned to the color number `-1`. After calling this function, `init_pair(x, curses.COLOR_RED, -1)` initializes, for instance, color pair *x* to a red foreground color on the default background.

`curses.wrapper(func, /, *args, **kwargs)`

Initialize curses and call another callable object, *func*, which should be the rest of your curses-using application. If the application raises an exception, this function will restore the terminal to a sane state before re-raising the exception and generating a traceback. The callable object *func* is then passed the main window 'stdscr' as its first argument, followed by any other arguments passed to `wrapper()`. Before calling *func*, `wrapper()` turns

on cbreak mode, turns off echo, enables the terminal keypad, and initializes colors if the terminal has color support. On exit (whether normally or by exception) it restores cooked mode, turns on echo, and disables the terminal keypad.

17.5.2 Window Objects

Window objects, as returned by `initscr()` and `newwin()` above, have the following methods and attributes :

`window.addch(ch[, attr])`

`window.addch(y, x, ch[, attr])`

Paint character *ch* at `(y, x)` with attributes *attr*, overwriting any character previously painted at that location. By default, the character position and attributes are the current settings for the window object.

> **Note**
>
> Writing outside the window, subwindow, or pad raises a `curses.error`. Attempting to write to the lower right corner of a window, subwindow, or pad will cause an exception to be raised after the character is printed.

`window.addnstr(str, n[, attr])`

`window.addnstr(y, x, str, n[, attr])`

Paint at most *n* characters of the character string *str* at `(y, x)` with attributes *attr*, overwriting anything previously on the display.

`window.addstr(str[, attr])`

`window.addstr(y, x, str[, attr])`

Paint the character string *str* at `(y, x)` with attributes *attr*, overwriting anything previously on the display.

> **Note**
>
> — Writing outside the window, subwindow, or pad raises `curses.error`. Attempting to write to the lower right corner of a window, subwindow, or pad will cause an exception to be raised after the string is printed.
> — A bug in ncurses, the backend for this Python module, can cause SegFaults when resizing windows. This is fixed in ncurses-6.1-20190511. If you are stuck with an earlier ncurses, you can avoid triggering this if you do not call `addstr()` with a *str* that has embedded newlines. Instead, call `addstr()` separately for each line.

`window.attroff(attr)`

Remove attribute *attr* from the "background" set applied to all writes to the current window.

`window.attron(attr)`

Add attribute *attr* from the "background" set applied to all writes to the current window.

`window.attrset(attr)`

Set the "background" set of attributes to *attr*. This set is initially `0` (no attributes).

`window.bkgd(ch[, attr])`

Set the background property of the window to the character *ch*, with attributes *attr*. The change is then applied to every character position in that window :

— The attribute of every character in the window is changed to the new background attribute.
— Wherever the former background character appears, it is changed to the new background character.

`window.bkgdset(ch[, attr])`

Set the window's background. A window's background consists of a character and any combination of attributes. The attribute part of the background is combined (OR'ed) with all non-blank characters that are written into

the window. Both the character and attribute parts of the background are combined with the blank characters. The background becomes a property of the character and moves with the character through any scrolling and insert/delete line/character operations.

`window.**border**([*ls*[, *rs*[, *ts*[, *bs*[, *tl*[, *tr*[, *bl*[, *br*]]]]]]]])`

Draw a border around the edges of the window. Each parameter specifies the character to use for a specific part of the border ; see the table below for more details.

> **Note**
>
> A `0` value for any parameter will cause the default character to be used for that parameter. Keyword parameters can *not* be used. The defaults are listed in this table :

Paramètre	Description	Valeur par défaut
ls	Left side	`ACS_VLINE`
rs	Right side	`ACS_VLINE`
ts	Top	`ACS_HLINE`
bs	Bottom	`ACS_HLINE`
tl	Upper-left corner	`ACS_ULCORNER`
tr	Upper-right corner	`ACS_URCORNER`
bl	Bottom-left corner	`ACS_LLCORNER`
br	Bottom-right corner	`ACS_LRCORNER`

`window.**box**([*vertch*, *horch*])`

Similar to `border()`, but both *ls* and *rs* are *vertch* and both *ts* and *bs* are *horch*. The default corner characters are always used by this function.

`window.**chgat**(*attr*)`

`window.**chgat**(*num*, *attr*)`

`window.**chgat**(*y*, *x*, *attr*)`

`window.**chgat**(*y*, *x*, *num*, *attr*)`

Set the attributes of *num* characters at the current cursor position, or at position `(y, x)` if supplied. If *num* is not given or is `-1`, the attribute will be set on all the characters to the end of the line. This function moves cursor to position `(y, x)` if supplied. The changed line will be touched using the `touchline()` method so that the contents will be redisplayed by the next window refresh.

`window.**clear**()`

Like `erase()`, but also cause the whole window to be repainted upon next call to `refresh()`.

`window.**clearok**(*flag*)`

If *flag* is `True`, the next call to `refresh()` will clear the window completely.

`window.**clrtobot**()`

Erase from cursor to the end of the window : all lines below the cursor are deleted, and then the equivalent of `clrtoeol()` is performed.

`window.**clrtoeol**()`

Erase from cursor to the end of the line.

`window.**cursyncup**()`

Update the current cursor position of all the ancestors of the window to reflect the current cursor position of the window.

`window.**delch**([*y*, *x*])`

Delete any character at `(y, x)`.

window.**deleteln**()

Delete the line under the cursor. All following lines are moved up by one line.

window.**derwin**(*begin_y*, *begin_x*)

window.**derwin**(*nlines*, *ncols*, *begin_y*, *begin_x*)

An abbreviation for "derive window", `derwin()` is the same as calling `subwin()`, except that *begin_y* and *begin_x* are relative to the origin of the window, rather than relative to the entire screen. Return a window object for the derived window.

window.**echochar**(*ch*[, *attr*])

Add character *ch* with attribute *attr*, and immediately call `refresh()` on the window.

window.**enclose**(*y*, *x*)

Test whether the given pair of screen-relative character-cell coordinates are enclosed by the given window, returning `True` or `False`. It is useful for determining what subset of the screen windows enclose the location of a mouse event.

Modifié dans la version 3.10 : Previously it returned `1` or `0` instead of `True` or `False`.

window.**encoding**

Encoding used to encode method arguments (Unicode strings and characters). The encoding attribute is inherited from the parent window when a subwindow is created, for example with `window.subwin()`. By default, current locale encoding is used (see `locale.getencoding()`).

Ajouté dans la version 3.3.

window.**erase**()

Clear the window.

window.**getbegyx**()

Return a tuple `(y, x)` of coordinates of upper-left corner.

window.**getbkgd**()

Return the given window's current background character/attribute pair.

window.**getch**([*y*, *x*])

Get a character. Note that the integer returned does *not* have to be in ASCII range : function keys, keypad keys and so on are represented by numbers higher than 255. In no-delay mode, return `-1` if there is no input, otherwise wait until a key is pressed.

window.**get_wch**([*y*, *x*])

Get a wide character. Return a character for most keys, or an integer for function keys, keypad keys, and other special keys. In no-delay mode, raise an exception if there is no input.

Ajouté dans la version 3.3.

window.**getkey**([*y*, *x*])

Get a character, returning a string instead of an integer, as `getch()` does. Function keys, keypad keys and other special keys return a multibyte string containing the key name. In no-delay mode, raise an exception if there is no input.

window.**getmaxyx**()

Return a tuple `(y, x)` of the height and width of the window.

window.**getparyx**()

Return the beginning coordinates of this window relative to its parent window as a tuple `(y, x)`. Return `(-1, -1)` if this window has no parent.

window.**getstr**()

window.**getstr**(*n*)

window.**getstr**(*y*, *x*)

window.**getstr**(*y*, *x*, *n*)

Read a bytes object from the user, with primitive line editing capacity.

window.**getyx**()

Return a tuple `(y, x)` of current cursor position relative to the window's upper-left corner.

window.**hline**(*ch*, *n*)

window.**hline**(*y*, *x*, *ch*, *n*)

Display a horizontal line starting at `(y, x)` with length *n* consisting of the character *ch*.

window.**idcok**(*flag*)

If *flag* is `False`, curses no longer considers using the hardware insert/delete character feature of the terminal ; if *flag* is `True`, use of character insertion and deletion is enabled. When curses is first initialized, use of character insert/delete is enabled by default.

window.**idlok**(*flag*)

If *flag* is `True`, *curses* will try and use hardware line editing facilities. Otherwise, line insertion/deletion are disabled.

window.**immedok**(*flag*)

If *flag* is `True`, any change in the window image automatically causes the window to be refreshed ; you no longer have to call *refresh()* yourself. However, it may degrade performance considerably, due to repeated calls to wrefresh. This option is disabled by default.

window.**inch**([*y*, *x*])

Return the character at the given position in the window. The bottom 8 bits are the character proper, and upper bits are the attributes.

window.**insch**(*ch*[, *attr*])

window.**insch**(*y*, *x*, *ch*[, *attr*])

Paint character *ch* at `(y, x)` with attributes *attr*, moving the line from position *x* right by one character.

window.**insdelln**(*nlines*)

Insert *nlines* lines into the specified window above the current line. The *nlines* bottom lines are lost. For negative *nlines*, delete *nlines* lines starting with the one under the cursor, and move the remaining lines up. The bottom *nlines* lines are cleared. The current cursor position remains the same.

window.**insertln**()

Insert a blank line under the cursor. All following lines are moved down by one line.

window.**insnstr**(*str*, *n*[, *attr*])

window.**insnstr**(*y*, *x*, *str*, *n*[, *attr*])

Insert a character string (as many characters as will fit on the line) before the character under the cursor, up to *n* characters. If *n* is zero or negative, the entire string is inserted. All characters to the right of the cursor are shifted right, with the rightmost characters on the line being lost. The cursor position does not change (after moving to *y*, *x*, if specified).

window.**insstr**(*str*[, *attr*])

window.**insstr**(*y*, *x*, *str*[, *attr*])

Insert a character string (as many characters as will fit on the line) before the character under the cursor. All characters to the right of the cursor are shifted right, with the rightmost characters on the line being lost. The cursor position does not change (after moving to *y*, *x*, if specified).

window.**instr**([*n*])

window.**instr**(*y*, *x*[, *n*])

Return a bytes object of characters, extracted from the window starting at the current cursor position, or at *y*, *x* if specified. Attributes are stripped from the characters. If *n* is specified, *instr()* returns a string at most *n* characters long (exclusive of the trailing NUL).

window.**is_linetouched**(*line*)

Return `True` if the specified line was modified since the last call to `refresh()`; otherwise return `False`. Raise a `curses.error` exception if *line* is not valid for the given window.

window.**is_wintouched**()

Return `True` if the specified window was modified since the last call to `refresh()`; otherwise return `False`.

window.**keypad**(*flag*)

If *flag* is `True`, escape sequences generated by some keys (keypad, function keys) will be interpreted by `curses`. If *flag* is `False`, escape sequences will be left as is in the input stream.

window.**leaveok**(*flag*)

If *flag* is `True`, cursor is left where it is on update, instead of being at "cursor position." This reduces cursor movement where possible. If possible the cursor will be made invisible.

If *flag* is `False`, cursor will always be at "cursor position" after an update.

window.**move**(*new_y*, *new_x*)

Move cursor to `(new_y, new_x)`.

window.**mvderwin**(*y*, *x*)

Move the window inside its parent window. The screen-relative parameters of the window are not changed. This routine is used to display different parts of the parent window at the same physical position on the screen.

window.**mvwin**(*new_y*, *new_x*)

Move the window so its upper-left corner is at `(new_y, new_x)`.

window.**nodelay**(*flag*)

If *flag* is `True`, `getch()` will be non-blocking.

window.**notimeout**(*flag*)

If *flag* is `True`, escape sequences will not be timed out.

If *flag* is `False`, after a few milliseconds, an escape sequence will not be interpreted, and will be left in the input stream as is.

window.**noutrefresh**()

Mark for refresh but wait. This function updates the data structure representing the desired state of the window, but does not force an update of the physical screen. To accomplish that, call `doupdate()`.

window.**overlay**(*destwin*[, *sminrow*, *smincol*, *dminrow*, *dmincol*, *dmaxrow*, *dmaxcol*])

Overlay the window on top of *destwin*. The windows need not be the same size, only the overlapping region is copied. This copy is non-destructive, which means that the current background character does not overwrite the old contents of *destwin*.

To get fine-grained control over the copied region, the second form of `overlay()` can be used. *sminrow* and *smincol* are the upper-left coordinates of the source window, and the other variables mark a rectangle in the destination window.

window.**overwrite**(*destwin*[, *sminrow*, *smincol*, *dminrow*, *dmincol*, *dmaxrow*, *dmaxcol*])

Overwrite the window on top of *destwin*. The windows need not be the same size, in which case only the overlapping region is copied. This copy is destructive, which means that the current background character overwrites the old contents of *destwin*.

To get fine-grained control over the copied region, the second form of `overwrite()` can be used. *sminrow* and *smincol* are the upper-left coordinates of the source window, the other variables mark a rectangle in the destination window.

window.**putwin**(*file*)

Write all data associated with the window into the provided file object. This information can be later retrieved using the `getwin()` function.

window.**redrawln**(*beg*, *num*)

Indicate that the *num* screen lines, starting at line *beg*, are corrupted and should be completely redrawn on the next `refresh()` call.

window.**redrawwin**()

Touch the entire window, causing it to be completely redrawn on the next `refresh()` call.

window.**refresh**([*pminrow*, *pmincol*, *sminrow*, *smincol*, *smaxrow*, *smaxcol*])

Update the display immediately (sync actual screen with previous drawing/deleting methods).

The 6 optional arguments can only be specified when the window is a pad created with `newpad()`. The additional parameters are needed to indicate what part of the pad and screen are involved. *pminrow* and *pmincol* specify the upper left-hand corner of the rectangle to be displayed in the pad. *sminrow*, *smincol*, *smaxrow*, and *smaxcol* specify the edges of the rectangle to be displayed on the screen. The lower right-hand corner of the rectangle to be displayed in the pad is calculated from the screen coordinates, since the rectangles must be the same size. Both rectangles must be entirely contained within their respective structures. Negative values of *pminrow*, *pmincol*, *sminrow*, or *smincol* are treated as if they were zero.

window.**resize**(*nlines*, *ncols*)

Reallocate storage for a curses window to adjust its dimensions to the specified values. If either dimension is larger than the current values, the window's data is filled with blanks that have the current background rendition (as set by `bkgdset()`) merged into them.

window.**scroll**([*lines=1*])

Scroll the screen or scrolling region upward by *lines* lines.

window.**scrollok**(*flag*)

Control what happens when the cursor of a window is moved off the edge of the window or scrolling region, either as a result of a newline action on the bottom line, or typing the last character of the last line. If *flag* is `False`, the cursor is left on the bottom line. If *flag* is `True`, the window is scrolled up one line. Note that in order to get the physical scrolling effect on the terminal, it is also necessary to call `idlok()`.

window.**setscrreg**(*top*, *bottom*)

Set the scrolling region from line *top* to line *bottom*. All scrolling actions will take place in this region.

window.**standend**()

Turn off the standout attribute. On some terminals this has the side effect of turning off all attributes.

window.**standout**()

Turn on attribute *A_STANDOUT*.

window.**subpad**(*begin_y*, *begin_x*)

window.**subpad**(*nlines*, *ncols*, *begin_y*, *begin_x*)

Return a sub-window, whose upper-left corner is at `(begin_y, begin_x)`, and whose width/height is *ncols*/*nlines*.

window.**subwin**(*begin_y*, *begin_x*)

window.**subwin**(*nlines*, *ncols*, *begin_y*, *begin_x*)

Return a sub-window, whose upper-left corner is at `(begin_y, begin_x)`, and whose width/height is *ncols*/*nlines*.

By default, the sub-window will extend from the specified position to the lower right corner of the window.

window.**syncdown**()

Touch each location in the window that has been touched in any of its ancestor windows. This routine is called by `refresh()`, so it should almost never be necessary to call it manually.

window.**syncok**(*flag*)

If *flag* is `True`, then `syncup()` is called automatically whenever there is a change in the window.

window.**syncup**()

Touch all locations in ancestors of the window that have been changed in the window.

window.**timeout**(*delay*)

Set blocking or non-blocking read behavior for the window. If *delay* is negative, blocking read is used (which will wait indefinitely for input). If *delay* is zero, then non-blocking read is used, and `getch()` will return `-1` if no input is waiting. If *delay* is positive, then `getch()` will block for *delay* milliseconds, and return `-1` if there is still no input at the end of that time.

window.**touchline**(*start*, *count*[, *changed*])

Pretend *count* lines have been changed, starting with line *start*. If *changed* is supplied, it specifies whether the affected lines are marked as having been changed (*changed*=`True`) or unchanged (*changed*=`False`).

window.**touchwin**()

Pretend the whole window has been changed, for purposes of drawing optimizations.

window.**untouchwin**()

Mark all lines in the window as unchanged since the last call to `refresh()`.

window.**vline**(*ch*, *n*[, *attr*])

window.**vline**(*y*, *x*, *ch*, *n*[, *attr*])

Display a vertical line starting at `(y, x)` with length *n* consisting of the character *ch* with attributes *attr*.

17.5.3 Constantes

The `curses` module defines the following data members :

curses.**ERR**

Some curses routines that return an integer, such as `getch()`, return `ERR` upon failure.

curses.**OK**

Some curses routines that return an integer, such as `napms()`, return `OK` upon success.

curses.**version**

curses.**__version__**

A bytes object representing the current version of the module.

curses.**ncurses_version**

A named tuple containing the three components of the ncurses library version : *major*, *minor*, and *patch*. All values are integers. The components can also be accessed by name, so `curses.ncurses_version[0]` is equivalent to `curses.ncurses_version.major` and so on.

Availability : if the ncurses library is used.

Ajouté dans la version 3.8.

curses.**COLORS**

The maximum number of colors the terminal can support. It is defined only after the call to `start_color()`.

curses.**COLOR_PAIRS**

The maximum number of color pairs the terminal can support. It is defined only after the call to `start_color()`.

curses.**COLS**

The width of the screen, i.e., the number of columns. It is defined only after the call to `initscr()`. Updated by `update_lines_cols()`, `resizeterm()` and `resize_term()`.

curses.**LINES**

The height of the screen, i.e., the number of lines. It is defined only after the call to `initscr()`. Updated by `update_lines_cols()`, `resizeterm()` and `resize_term()`.

Some constants are available to specify character cell attributes. The exact constants available are system dependent.

Attribut	Signification
curses.**A_ALTCHARSET**	Alternate character set mode
curses.**A_BLINK**	Blink mode
curses.**A_BOLD**	Bold mode
curses.**A_DIM**	Dim mode
curses.**A_INVIS**	Invisible or blank mode
curses.**A_ITALIC**	Italic mode
curses.**A_NORMAL**	Attribut normal
curses.**A_PROTECT**	Protected mode
curses.**A_REVERSE**	Reverse background and foreground colors
curses.**A_STANDOUT**	Standout mode
curses.**A_UNDERLINE**	Underline mode
curses.**A_HORIZONTAL**	Horizontal highlight
curses.**A_LEFT**	Left highlight
curses.**A_LOW**	Low highlight
curses.**A_RIGHT**	Right highlight
curses.**A_TOP**	Top highlight
curses.**A_VERTICAL**	Vertical highlight

Ajouté dans la version 3.7 : `A_ITALIC` was added.

Several constants are available to extract corresponding attributes returned by some methods.

Bit-mask	Signification
curses.**A_ATTRIBUTES**	Bit-mask to extract attributes
curses.**A_CHARTEXT**	Bit-mask to extract a character
curses.**A_COLOR**	Bit-mask to extract color-pair field information

Keys are referred to by integer constants with names starting with `KEY_`. The exact keycaps available are system dependent.

Key constant	Clé
curses.**KEY_MIN**	Minimum key value
curses.**KEY_BREAK**	Break key (unreliable)
curses.**KEY_DOWN**	Down-arrow
curses.**KEY_UP**	Up-arrow
curses.**KEY_LEFT**	Left-arrow
curses.**KEY_RIGHT**	Right-arrow
curses.**KEY_HOME**	Home key (upward+left arrow)
curses.**KEY_BACKSPACE**	Backspace (unreliable)
curses.**KEY_F0**	Function keys. Up to 64 function keys are supported.
curses.**KEY_Fn**	Value of function key *n*
curses.**KEY_DL**	Delete line
curses.**KEY_IL**	Insert line
curses.**KEY_DC**	Delete character

suite sur la page suivante

Tableau 1 – suite de la page précédente

Key constant	Clé
curses.**KEY_IC**	Insert char or enter insert mode
curses.**KEY_EIC**	Exit insert char mode
curses.**KEY_CLEAR**	Clear screen
curses.**KEY_EOS**	Clear to end of screen
curses.**KEY_EOL**	Clear to end of line
curses.**KEY_SF**	Scroll 1 line forward
curses.**KEY_SR**	Scroll 1 line backward (reverse)
curses.**KEY_NPAGE**	Next page
curses.**KEY_PPAGE**	Previous page
curses.**KEY_STAB**	Set tab
curses.**KEY_CTAB**	Clear tab
curses.**KEY_CATAB**	Clear all tabs
curses.**KEY_ENTER**	Enter or send (unreliable)
curses.**KEY_SRESET**	Soft (partial) reset (unreliable)
curses.**KEY_RESET**	Reset or hard reset (unreliable)
curses.**KEY_PRINT**	Print
curses.**KEY_LL**	Home down or bottom (lower left)
curses.**KEY_A1**	Upper left of keypad

suite sur la page suivante

Tableau 1 – suite de la page précédente

Key constant	Clé
curses.**KEY_A3**	Upper right of keypad
curses.**KEY_B2**	Center of keypad
curses.**KEY_C1**	Lower left of keypad
curses.**KEY_C3**	Lower right of keypad
curses.**KEY_BTAB**	Back tab
curses.**KEY_BEG**	Beg (beginning)
curses.**KEY_CANCEL**	Cancel
curses.**KEY_CLOSE**	*Close*
curses.**KEY_COMMAND**	Cmd (command)
curses.**KEY_COPY**	*Copy*
curses.**KEY_CREATE**	Create
curses.**KEY_END**	End
curses.**KEY_EXIT**	*Exit*
curses.**KEY_FIND**	Find
curses.**KEY_HELP**	Help
curses.**KEY_MARK**	Mark
curses.**KEY_MESSAGE**	Message
curses.**KEY_MOVE**	Move

suite sur la page suivante

Tableau 1 – suite de la page précédente

Key constant	Clé
curses.**KEY_NEXT**	Next
curses.**KEY_OPEN**	Open
curses.**KEY_OPTIONS**	Options
curses.**KEY_PREVIOUS**	Prev (previous)
curses.**KEY_REDO**	*Redo*
curses.**KEY_REFERENCE**	Ref (reference)
curses.**KEY_REFRESH**	Refresh
curses.**KEY_REPLACE**	Replace
curses.**KEY_RESTART**	Restart
curses.**KEY_RESUME**	Resume
curses.**KEY_SAVE**	*Save*
curses.**KEY_SBEG**	Shifted Beg (beginning)
curses.**KEY_SCANCEL**	Shifted Cancel
curses.**KEY_SCOMMAND**	Shifted Command
curses.**KEY_SCOPY**	Shifted Copy
curses.**KEY_SCREATE**	Shifted Create
curses.**KEY_SDC**	Shifted Delete char
curses.**KEY_SDL**	Shifted Delete line

suite sur la page suivante

Tableau 1 – suite de la page précédente

Key constant	Clé
curses.**KEY_SELECT**	Select
curses.**KEY_SEND**	Shifted End
curses.**KEY_SEOL**	Shifted Clear line
curses.**KEY_SEXIT**	Shifted Exit
curses.**KEY_SFIND**	Shifted Find
curses.**KEY_SHELP**	Shifted Help
curses.**KEY_SHOME**	Shifted Home
curses.**KEY_SIC**	Shifted Input
curses.**KEY_SLEFT**	Shifted Left arrow
curses.**KEY_SMESSAGE**	Shifted Message
curses.**KEY_SMOVE**	Shifted Move
curses.**KEY_SNEXT**	Shifted Next
curses.**KEY_SOPTIONS**	Shifted Options
curses.**KEY_SPREVIOUS**	Shifted Prev
curses.**KEY_SPRINT**	Shifted Print
curses.**KEY_SREDO**	Shifted Redo
curses.**KEY_SREPLACE**	Shifted Replace
curses.**KEY_SRIGHT**	Shifted Right arrow

suite sur la page suivante

Tableau 1 – suite de la page précédente

Key constant	Clé
curses.**KEY_SRSUME**	Shifted Resume
curses.**KEY_SSAVE**	Shifted Save
curses.**KEY_SSUSPEND**	Shifted Suspend
curses.**KEY_SUNDO**	Shifted Undo
curses.**KEY_SUSPEND**	Suspend
curses.**KEY_UNDO**	*Undo*
curses.**KEY_MOUSE**	Mouse event has occurred
curses.**KEY_RESIZE**	Terminal resize event
curses.**KEY_MAX**	Maximum key value

On VT100s and their software emulations, such as X terminal emulators, there are normally at least four function keys (*KEY_F1*, *KEY_F2*, *KEY_F3*, *KEY_F4*) available, and the arrow keys mapped to *KEY_UP*, *KEY_DOWN*, *KEY_LEFT* and *KEY_RIGHT* in the obvious way. If your machine has a PC keyboard, it is safe to expect arrow keys and twelve function keys (older PC keyboards may have only ten function keys) ; also, the following keypad mappings are standard :

Keycap	Constante
Insert	KEY_IC
Delete	KEY_DC
Home	KEY_HOME
End	KEY_END
Page Up	KEY_PPAGE
Page Down	KEY_NPAGE

The following table lists characters from the alternate character set. These are inherited from the VT100 terminal, and will generally be available on software emulations such as X terminals. When there is no graphic available, curses falls back on a crude printable ASCII approximation.

Note

These are available only after *initscr()* has been called.

ACS code	Signification
curses.**ACS_BBSS**	alternate name for upper right corner
curses.**ACS_BLOCK**	solid square block
curses.**ACS_BOARD**	board of squares
curses.**ACS_BSBS**	alternate name for horizontal line
curses.**ACS_BSSB**	alternate name for upper left corner
curses.**ACS_BSSS**	alternate name for top tee
curses.**ACS_BTEE**	bottom tee
curses.**ACS_BULLET**	bullet
curses.**ACS_CKBOARD**	checker board (stipple)
curses.**ACS_DARROW**	arrow pointing down
curses.**ACS_DEGREE**	degree symbol
curses.**ACS_DIAMOND**	diamond
curses.**ACS_GEQUAL**	greater-than-or-equal-to
curses.**ACS_HLINE**	horizontal line
curses.**ACS_LANTERN**	lantern symbol
curses.**ACS_LARROW**	left arrow
curses.**ACS_LEQUAL**	less-than-or-equal-to
curses.**ACS_LLCORNER**	lower left-hand corner

suite sur la page suivante

Tableau 2 – suite de la page précédente

ACS code	Signification
curses.**ACS_LRCORNER**	lower right-hand corner
curses.**ACS_LTEE**	left tee
curses.**ACS_NEQUAL**	not-equal sign
curses.**ACS_PI**	letter pi
curses.**ACS_PLMINUS**	plus-or-minus sign
curses.**ACS_PLUS**	big plus sign
curses.**ACS_RARROW**	right arrow
curses.**ACS_RTEE**	right tee
curses.**ACS_S1**	scan line 1
curses.**ACS_S3**	scan line 3
curses.**ACS_S7**	scan line 7
curses.**ACS_S9**	scan line 9
curses.**ACS_SBBS**	alternate name for lower right corner
curses.**ACS_SBSB**	alternate name for vertical line
curses.**ACS_SBSS**	alternate name for right tee
curses.**ACS_SSBB**	alternate name for lower left corner
curses.**ACS_SSBS**	alternate name for bottom tee
curses.**ACS_SSSB**	alternate name for left tee

suite sur la page suivante

Tableau 2 – suite de la page précédente

ACS code	Signification
curses.**ACS_SSSS**	alternate name for crossover or big plus
curses.**ACS_STERLING**	pound sterling
curses.**ACS_TTEE**	top tee
curses.**ACS_UARROW**	up arrow
curses.**ACS_ULCORNER**	upper left corner
curses.**ACS_URCORNER**	upper right corner
curses.**ACS_VLINE**	vertical line

The following table lists mouse button constants used by *getmouse()* :

Mouse button constant	Signification
curses.**BUTTONn_PRESSED**	Mouse button *n* pressed
curses.**BUTTONn_RELEASED**	Mouse button *n* released
curses.**BUTTONn_CLICKED**	Mouse button *n* clicked
curses.**BUTTONn_DOUBLE_CLICKED**	Mouse button *n* double clicked
curses.**BUTTONn_TRIPLE_CLICKED**	Mouse button *n* triple clicked
curses.**BUTTON_SHIFT**	Shift was down during button state change
curses.**BUTTON_CTRL**	Control was down during button state change
curses.**BUTTON_ALT**	Control was down during button state change

Modifié dans la version 3.10 : The `BUTTON5_*` constants are now exposed if they are provided by the underlying curses library.

The following table lists the predefined colors :

Constante	Color
curses.**COLOR_BLACK**	Black
curses.**COLOR_BLUE**	Blue
curses.**COLOR_CYAN**	Cyan (light greenish blue)
curses.**COLOR_GREEN**	Green
curses.**COLOR_MAGENTA**	Magenta (purplish red)
curses.**COLOR_RED**	Red
curses.**COLOR_WHITE**	White
curses.**COLOR_YELLOW**	Yellow

17.6 curses.textpad --- Text input widget for curses programs

The *curses.textpad* module provides a *Textbox* class that handles elementary text editing in a curses window, supporting a set of keybindings resembling those of Emacs (thus, also of Netscape Navigator, BBedit 6.x, FrameMaker, and many other programs). The module also provides a rectangle-drawing function useful for framing text boxes or for other purposes.

The module *curses.textpad* defines the following function :

curses.textpad.**rectangle**(*win*, *uly*, *ulx*, *lry*, *lrx*)

Draw a rectangle. The first argument must be a window object; the remaining arguments are coordinates relative to that window. The second and third arguments are the y and x coordinates of the upper left hand corner of the rectangle to be drawn; the fourth and fifth arguments are the y and x coordinates of the lower right hand corner. The rectangle will be drawn using VT100/IBM PC forms characters on terminals that make this possible (including xterm and most other software terminal emulators). Otherwise it will be drawn with ASCII dashes, vertical bars, and plus signs.

17.6.1 Textbox objects

You can instantiate a *Textbox* object as follows :

class curses.textpad.**Textbox**(*win*)

Return a textbox widget object. The *win* argument should be a curses *window* object in which the textbox is to be contained. The edit cursor of the textbox is initially located at the upper left hand corner of the containing window, with coordinates `(0, 0)`. The instance's *stripspaces* flag is initially on.

Textbox objects have the following methods :

edit([*validator*])

This is the entry point you will normally use. It accepts editing keystrokes until one of the termination keystrokes is entered. If *validator* is supplied, it must be a function. It will be called for each keystroke

entered with the keystroke as a parameter ; command dispatch is done on the result. This method returns the window contents as a string ; whether blanks in the window are included is affected by the `stripspaces` attribute.

do_command(*ch*)

Process a single command keystroke. Here are the supported special keystrokes :

Keystroke	Action
`Control-A`	Go to left edge of window.
`Control-B`	Cursor left, wrapping to previous line if appropriate.
`Control-D`	Delete character under cursor.
`Control-E`	Go to right edge (stripspaces off) or end of line (stripspaces on).
`Control-F`	Cursor right, wrapping to next line when appropriate.
`Control-G`	Terminate, returning the window contents.
`Control-H`	Delete character backward.
`Control-J`	Terminate if the window is 1 line, otherwise insert newline.
`Control-K`	If line is blank, delete it, otherwise clear to end of line.
`Control-L`	Refresh screen.
`Control-N`	Cursor down ; move down one line.
`Control-O`	Insert a blank line at cursor location.
`Control-P`	Cursor up ; move up one line.

Move operations do nothing if the cursor is at an edge where the movement is not possible. The following synonyms are supported where possible :

Constante	Keystroke
KEY_LEFT	`Control-B`
KEY_RIGHT	`Control-F`
KEY_UP	`Control-P`
KEY_DOWN	`Control-N`
KEY_BACKSPACE	`Control-h`

All other keystrokes are treated as a command to insert the given character and move right (with line wrapping).

gather()

Return the window contents as a string ; whether blanks in the window are included is affected by the `stripspaces` member.

stripspaces

This attribute is a flag which controls the interpretation of blanks in the window. When it is on, trailing blanks on each line are ignored ; any cursor motion that would land the cursor on a trailing blank goes to the end of that line instead, and trailing blanks are stripped when the window contents are gathered.

17.7 `curses.ascii` --- Utilities for ASCII characters

Source code : Lib/curses/ascii.py

The `curses.ascii` module supplies name constants for ASCII characters and functions to test membership in various ASCII character classes. The constants supplied are names for control characters as follows :

Nom	Signification
curses.ascii.**NUL**	
curses.ascii.**SOH**	Start of heading, console interrupt
curses.ascii.**STX**	Start of text
curses.ascii.**ETX**	End of text
curses.ascii.**EOT**	End of transmission
curses.ascii.**ENQ**	Enquiry, goes with *ACK* flow control
curses.ascii.**ACK**	Acknowledgement
curses.ascii.**BEL**	Bell
curses.ascii.**BS**	Backspace
curses.ascii.**TAB**	Tab
curses.ascii.**HT**	Alias for *TAB* : "Horizontal tab"
curses.ascii.**LF**	Line feed
curses.ascii.**NL**	Alias for *LF* : "New line"
curses.ascii.**VT**	Vertical tab
curses.ascii.**FF**	Form feed
curses.ascii.**CR**	Retour chariot
curses.ascii.**SO**	Shift-out, begin alternate character set
curses.ascii.**SI**	Shift-in, resume default character set

suite sur la page suivante

Tableau 3 – suite de la page précédente

Nom	Signification
curses.ascii.**DLE**	Data-link escape
curses.ascii.**DC1**	XON, for flow control
curses.ascii.**DC2**	Device control 2, block-mode flow control
curses.ascii.**DC3**	XOFF, for flow control
curses.ascii.**DC4**	Device control 4
curses.ascii.**NAK**	Negative acknowledgement
curses.ascii.**SYN**	Synchronous idle
curses.ascii.**ETB**	End transmission block
curses.ascii.**CAN**	Cancel
curses.ascii.**EM**	End of medium
curses.ascii.**SUB**	Substitute
curses.ascii.**ESC**	Escape
curses.ascii.**FS**	Séparateur de fichiers
curses.ascii.**GS**	Séparateur de groupe
curses.ascii.**RS**	Record separator, block-mode terminator
curses.ascii.**US**	Unit separator
curses.ascii.**SP**	Space
curses.ascii.**DEL**	Delete

Note that many of these have little practical significance in modern usage. The mnemonics derive from teleprinter

conventions that predate digital computers.

The module supplies the following functions, patterned on those in the standard C library :

`curses.ascii.isalnum(c)`

Checks for an ASCII alphanumeric character ; it is equivalent to `isalpha(c) or isdigit(c)`.

`curses.ascii.isalpha(c)`

Checks for an ASCII alphabetic character ; it is equivalent to `isupper(c) or islower(c)`.

`curses.ascii.isascii(c)`

Checks for a character value that fits in the 7-bit ASCII set.

`curses.ascii.isblank(c)`

Checks for an ASCII whitespace character ; space or horizontal tab.

`curses.ascii.iscntrl(c)`

Checks for an ASCII control character (in the range 0x00 to 0x1f or 0x7f).

`curses.ascii.isdigit(c)`

Checks for an ASCII decimal digit, `'0'` through `'9'`. This is equivalent to `c in string.digits`.

`curses.ascii.isgraph(c)`

Checks for ASCII any printable character except space.

`curses.ascii.islower(c)`

Checks for an ASCII lower-case character.

`curses.ascii.isprint(c)`

Checks for any ASCII printable character including space.

`curses.ascii.ispunct(c)`

Checks for any printable ASCII character which is not a space or an alphanumeric character.

`curses.ascii.isspace(c)`

Checks for ASCII white-space characters ; space, line feed, carriage return, form feed, horizontal tab, vertical tab.

`curses.ascii.isupper(c)`

Checks for an ASCII uppercase letter.

`curses.ascii.isxdigit(c)`

Checks for an ASCII hexadecimal digit. This is equivalent to `c in string.hexdigits`.

`curses.ascii.isctrl(c)`

Checks for an ASCII control character (ordinal values 0 to 31).

`curses.ascii.ismeta(c)`

Checks for a non-ASCII character (ordinal values 0x80 and above).

These functions accept either integers or single-character strings ; when the argument is a string, it is first converted using the built-in function `ord()`.

Note that all these functions check ordinal bit values derived from the character of the string you pass in ; they do not actually know anything about the host machine's character encoding.

The following two functions take either a single-character string or integer byte value ; they return a value of the same type.

`curses.ascii.ascii(c)`

Return the ASCII value corresponding to the low 7 bits of *c*.

`curses.ascii.ctrl(c)`

Return the control character corresponding to the given character (the character bit value is bitwise-anded with 0x1f).

`curses.ascii.alt(c)`

Return the 8-bit character corresponding to the given ASCII character (the character bit value is bitwise-ored with 0x80).

The following function takes either a single-character string or integer value ; it returns a string.

`curses.ascii.unctrl(c)`

Return a string representation of the ASCII character *c*. If *c* is printable, this string is the character itself. If the character is a control character (0x00--0x1f) the string consists of a caret (`'^'`) followed by the corresponding uppercase letter. If the character is an ASCII delete (0x7f) the string is `'^?'`. If the character has its meta bit (0x80) set, the meta bit is stripped, the preceding rules applied, and `'!'` prepended to the result.

`curses.ascii.controlnames`

A 33-element string array that contains the ASCII mnemonics for the thirty-two ASCII control characters from 0 (NUL) to 0x1f (US), in order, plus the mnemonic `SP` for the space character.

17.8 `curses.panel` --- A panel stack extension for curses

Panels are windows with the added feature of depth, so they can be stacked on top of each other, and only the visible portions of each window will be displayed. Panels can be added, moved up or down in the stack, and removed.

17.8.1 Fonctions

The module `curses.panel` defines the following functions :

`curses.panel.bottom_panel()`

Returns the bottom panel in the panel stack.

`curses.panel.new_panel(win)`

Returns a panel object, associating it with the given window *win*. Be aware that you need to keep the returned panel object referenced explicitly. If you don't, the panel object is garbage collected and removed from the panel stack.

`curses.panel.top_panel()`

Returns the top panel in the panel stack.

`curses.panel.update_panels()`

Updates the virtual screen after changes in the panel stack. This does not call `curses.doupdate()`, so you'll have to do this yourself.

17.8.2 Panel Objects

Panel objects, as returned by `new_panel()` above, are windows with a stacking order. There's always a window associated with a panel which determines the content, while the panel methods are responsible for the window's depth in the panel stack.

Panel objects have the following methods :

`Panel.above()`

Returns the panel above the current panel.

`Panel.below()`

Returns the panel below the current panel.

`Panel.bottom()`

Push the panel to the bottom of the stack.

`Panel.hidden()`

Returns `True` if the panel is hidden (not visible), `False` otherwise.

`Panel.hide()`

Hide the panel. This does not delete the object, it just makes the window on screen invisible.

`Panel.move(y, x)`

Move the panel to the screen coordinates `(y, x)`.

`Panel.replace(win)`

Change the window associated with the panel to the window *win*.

`Panel.set_userptr(obj)`

Set the panel's user pointer to *obj*. This is used to associate an arbitrary piece of data with the panel, and can be any Python object.

`Panel.show()`

Display the panel (which might have been hidden).

`Panel.top()`

Push panel to the top of the stack.

`Panel.userptr()`

Returns the user pointer for the panel. This might be any Python object.

`Panel.window()`

Returns the window object associated with the panel.

CHAPITRE 18

Exécution concourante

Les modules documentés dans ce chapitre fournissent des outils d'exécution concourante de code. Le choix de l'outil approprié dépend de la tâche à exécuter (limitée par le CPU (*CPU bound*), ou limitée la vitesse des entrées-sorties (*IO bound*)) et du style de développement désiré (coopération gérée par des évènements ou multitâche préemptif). En voici un survol :

18.1 `threading` --- Thread-based parallelism

Code source : Lib/threading.py

This module constructs higher-level threading interfaces on top of the lower level `_thread` module.

Modifié dans la version 3.7 : Ce module était auparavant optionnel, il est maintenant toujours disponible.

Voir aussi

`concurrent.futures.ThreadPoolExecutor` offers a higher level interface to push tasks to a background thread without blocking execution of the calling thread, while still being able to retrieve their results when needed.

`queue` provides a thread-safe interface for exchanging data between running threads.

`asyncio` offers an alternative approach to achieving task level concurrency without requiring the use of multiple operating system threads.

Note

In the Python 2.x series, this module contained `camelCase` names for some methods and functions. These are deprecated as of Python 3.10, but they are still supported for compatibility with Python 2.5 and lower.

Particularité de l'implémentation CPython : In CPython, due to the *Global Interpreter Lock*, only one thread can execute Python code at once (even though certain performance-oriented libraries might overcome this limitation). If you want your application to make better use of the computational resources of multi-core machines, you are advised to use `multiprocessing` or `concurrent.futures.ProcessPoolExecutor`. However, threading is still an appropriate model if you want to run multiple I/O-bound tasks simultaneously.

Availability : not WASI.

This module does not work or is not available on WebAssembly. See *Plateformes WebAssembly* for more information.

Ce module définit les fonctions suivantes :

threading.**active_count**()

Renvoie le nombre d'objets `Thread` actuellement vivants. Le compte renvoyé est égal à la longueur de la liste renvoyée par `enumerate()`.

The function `activeCount` is a deprecated alias for this function.

threading.**current_thread**()

Renvoie l'objet `Thread` courant, correspondant au fil de contrôle de l'appelant. Si le fil de contrôle de l'appelant n'a pas été créé via le module `Thread`, un objet *thread* factice aux fonctionnalités limitées est renvoyé.

The function `currentThread` is a deprecated alias for this function.

threading.**excepthook**(*args*, /)

Gère les exceptions non-attrapées levées par `Thread.run()`.

L'argument *arg* a les attributs suivants :
— *exc_type* : le type de l'exception ;
— *exc_value* : la valeur de l'exception, peut être `None` ;
— *exc_traceback* : la pile d'appels pour cette exception, peut être `None` ;
— *thread* : le fil d'exécution ayant levé l'exception, peut être `None`.
Si *exc_type* est `SystemExit`, l'exception est ignorée silencieusement. Toutes les autres sont affichées sur `sys.stderr`.

Si cette fonction lève une exception, `sys.excepthook()` est appelée pour la gérer.

La fonction `threading.excepthook()` peut être surchargée afin de contrôler comment les exceptions non-attrapées levées par `Thread.run()` sont gérées.

Stocker *exc_value* en utilisant une fonction de rappel personnalisée peut créer un cycle de références. *exc_value* doit être nettoyée explicitement pour casser ce cycle lorsque l'exception n'est plus nécessaire.

Stocker *thread* en utilisant une fonction de rappel personnalisée peut le ressusciter, si c'est un objet en cours de finalisation. Évitez de stocker *thread* après la fin de la fonction de rappel, pour éviter de ressusciter des objets.

Voir aussi

`sys.excepthook()` gère les exceptions qui n'ont pas été attrapées.

Ajouté dans la version 3.8.

threading.**__excepthook__**

Holds the original value of `threading.excepthook()`. It is saved so that the original value can be restored in case they happen to get replaced with broken or alternative objects.

Ajouté dans la version 3.10.

threading.**get_ident**()

Renvoie l'« identifiant de fil » du fil d'exécution courant. C'est un entier non nul. Sa valeur n'a pas de signification directe ; il est destiné à être utilisé comme valeur magique opaque, par exemple comme clef de dictionnaire de données pour chaque fil. Les identificateurs de fils peuvent être recyclés lorsqu'un fil se termine et qu'un autre fil est créé.

Ajouté dans la version 3.3.

threading.**get_native_id**()

Renvoie l'identifiant natif complet assigné par le noyau du fil d'exécution actuel. C'est un entier non négatif. Sa valeur peut uniquement être utilisée pour identifier ce fil d'exécution à l'échelle du système (jusqu'à ce que le fil d'exécution se termine, après quoi la valeur peut être recyclée par le système d'exploitation).

Availability : Windows, FreeBSD, Linux, macOS, OpenBSD, NetBSD, AIX, DragonFlyBSD, GNU/kFreeBSD.

Ajouté dans la version 3.8.

Modifié dans la version 3.13 : Added support for GNU/kFreeBSD.

threading.**enumerate**()

Return a list of all `Thread` objects currently active. The list includes daemonic threads and dummy thread objects created by `current_thread()`. It excludes terminated threads and threads that have not yet been started. However, the main thread is always part of the result, even when terminated.

threading.**main_thread**()

Renvoie l'objet fil d'exécution `Thread` principal. Dans des conditions normales, le fil principal est le fil à partir duquel l'interpréteur Python a été lancé.

Ajouté dans la version 3.4.

threading.**settrace**(*func*)

Attache une fonction de traçage pour tous les fils d'exécution démarrés depuis le module `Thread`. La fonction *func* est passée à `sys.settrace()` pour chaque fil, avant que sa méthode `run()` soit appelée.

threading.**settrace_all_threads**(*func*)

Set a trace function for all threads started from the `threading` module and all Python threads that are currently executing.

The *func* will be passed to `sys.settrace()` for each thread, before its `run()` method is called.

Ajouté dans la version 3.12.

threading.**gettrace**()

Renvoie la fonction de traçage tel que définie par `settrace()`.

Ajouté dans la version 3.10.

threading.**setprofile**(*func*)

Attache une fonction de profilage pour tous les fils d'exécution démarrés depuis le module `Threading`. La fonction *func* est passée à `sys.setprofile()` pour chaque fil, avant que sa méthode `run()` soit appelée.

threading.**setprofile_all_threads**(*func*)

Set a profile function for all threads started from the `threading` module and all Python threads that are currently executing.

The *func* will be passed to `sys.setprofile()` for each thread, before its `run()` method is called.

Ajouté dans la version 3.12.

threading.**getprofile**()

Renvoie la fonction de profilage tel que défini par `setprofile()`.

Ajouté dans la version 3.10.

threading.**stack_size**([*size*])

Renvoie la taille de la pile d'exécution utilisée lors de la création de nouveaux fils d'exécution. L'argument optionnel *size* spécifie la taille de pile à utiliser pour les fils créés ultérieurement, et doit être 0 (pour utiliser la taille de la plate-forme ou la valeur configurée par défaut) ou un entier positif supérieur ou égal à 32 768 (32 Kio). Si *size* n'est pas spécifié, 0 est utilisé. Si la modification de la taille de la pile de fils n'est pas prise en charge, une `RuntimeError` est levée. Si la taille de pile spécifiée n'est pas valide, une `ValueError` est levée et la taille de pile n'est pas modifiée. 32 Kio est actuellement la valeur minimale de taille de pile prise en charge pour garantir un espace de pile suffisant pour l'interpréteur lui-même. Notez que certaines plates-formes peuvent avoir des restrictions particulières sur les valeurs de taille de la pile, telles que l'exigence d'une taille de pile minimale > 32 Kio ou d'une allocation en multiples de la taille de page de la mémoire du système – la documentation de la plate-forme devrait être consultée pour plus d'informations (4 Kio sont courantes ; en l'absence de renseignements plus spécifiques, l'approche proposée est l'utilisation de multiples de 4 096 pour la taille de la pile).

Availability : Windows, pthreads.

Unix platforms with POSIX threads support.

Ce module définit également la constante suivante :

threading.**TIMEOUT_MAX**

La valeur maximale autorisée pour le paramètre *timeout* des fonctions bloquantes (`Lock.acquire()`, `RLock.acquire()`, `Condition.wait()`, etc.). Spécifier un délai d'attente supérieur à cette valeur lève une `OverflowError`.

Ajouté dans la version 3.2.

Ce module définit un certain nombre de classes, qui sont détaillées dans les sections ci-dessous.

La conception de ce module est librement basée sur le modèle des fils d'exécution de Java. Cependant, là où Java fait des verrous et des variables de condition le comportement de base de chaque objet, ils sont des objets séparés en Python. La classe Python `Thread` prend en charge un sous-ensemble du comportement de la classe *Thread* de Java ; actuellement, il n'y a aucune priorité, aucun groupe de fils d'exécution, et les fils ne peuvent être détruits, arrêtés, suspendus, repris ni interrompus. Les méthodes statiques de la classe *Thread* de Java, lorsqu'elles sont implémentées, correspondent à des fonctions au niveau du module.

Toutes les méthodes décrites ci-dessous sont exécutées de manière atomique.

18.1.1 Données locales au fil d'exécution

Les données locales au fil d'exécution (*thread-local data*) sont des données dont les valeurs sont propres à chaque fil. Pour gérer les données locales au fil, il suffit de créer une instance de `local` (ou une sous-classe) et d'y stocker des données :

```
mydata = threading.local()
mydata.x = 1
```

Les valeurs dans l'instance sont différentes pour des *threads* différents.

class threading.**local**

Classe qui représente les données locales au fil d'exécution.

For more details and extensive examples, see the documentation string of the `_threading_local` module : Lib/_threading_local.py.

18.1.2 Objets *Threads*

The `Thread` class represents an activity that is run in a separate thread of control. There are two ways to specify the activity : by passing a callable object to the constructor, or by overriding the `run()` method in a subclass. No other methods (except for the constructor) should be overridden in a subclass. In other words, *only* override the `__init__()` and `run()` methods of this class.

Une fois qu'un objet fil d'exécution est créé, son activité doit être lancée en appelant la méthode `start()` du fil. Ceci invoque la méthode `run()` dans un fil d'exécution séparé.

Une fois que l'activité du fil d'exécution est lancée, le fil est considéré comme « vivant ». Il cesse d'être vivant lorsque sa méthode `run()` se termine – soit normalement, soit en levant une exception non gérée. La méthode `is_alive()` teste si le fil est vivant.

D'autres fils d'exécution peuvent appeler la méthode `join()` d'un fil. Ceci bloque le fil appelant jusqu'à ce que le fil dont la méthode `join()` est appelée soit terminé.

Un fil d'exécution a un nom. Le nom peut être passé au constructeur, et lu ou modifié via l'attribut `name`.

Si la méthode `run()` lève une exception, `threading.excepthook()` est appelée pour s'en occuper. Par défaut, `threading.excepthook()` ignore silencieusement `SystemExit`.

Un fil d'exécution peut être marqué comme « fil démon ». Un programme Python se termine quand il ne reste plus que des fils démons. La valeur initiale est héritée du fil d'exécution qui l'a créé. Cette option peut être définie par la propriété `daemon` ou par l'argument *daemon* du constructeur.

> **Note**
>
> Les fils d'exécution démons sont brusquement terminés à l'arrêt du programme Python. Leurs ressources (fichiers ouverts, transactions de base de données, etc.) peuvent ne pas être libérées correctement. Si vous voulez que vos fils s'arrêtent proprement, faites en sorte qu'ils ne soient pas démoniques et utilisez un mécanisme de signalisation approprié tel qu'un objet évènement `Event`.

Il y a un objet "fil principal", qui correspond au fil de contrôle initial dans le programme Python. Ce n'est pas un fil démon.

There is the possibility that "dummy thread objects" are created. These are thread objects corresponding to "alien threads", which are threads of control started outside the threading module, such as directly from C code. Dummy thread objects have limited functionality ; they are always considered alive and daemonic, and cannot be *joined*. They are never deleted, since it is impossible to detect the termination of alien threads.

class `threading.`**`Thread`**(*group=None*, *target=None*, *name=None*, *args=()*, *kwargs={}*, *, *daemon=None*)

Ce constructeur doit toujours être appelé avec des arguments nommés. Les arguments sont :

group should be `None` ; reserved for future extension when a `ThreadGroup` class is implemented.

target est l'objet appelable qui doit être invoqué par la méthode `run()`. La valeur par défaut est `None`, ce qui signifie que rien n'est appelé.

name is the thread name. By default, a unique name is constructed of the form "Thread-*N*" where *N* is a small decimal number, or "Thread-*N* (target)" where "target" is `target.__name__` if the *target* argument is specified.

args is a list or tuple of arguments for the target invocation. Defaults to `()`.

kwargs est un dictionnaire d'arguments nommés pour l'invocation de l'objet appelable. La valeur par défaut est `{}`.

S'il ne vaut pas `None`, *daemon* définit explicitement si le fil d'exécution est démonique ou pas. S'il vaut `None` (par défaut), la valeur est héritée du fil courant.

Si la sous-classe réimplémente le constructeur, elle doit s'assurer d'appeler le constructeur de la classe de base (`Thread.__init__()`) avant de faire autre chose au fil d'exécution.

Modifié dans la version 3.3 : Added the *daemon* parameter.

Modifié dans la version 3.10 : Use the *target* name if *name* argument is omitted.

start()

Lance l'activité du fil d'exécution.

Elle ne doit être appelée qu'une fois par objet de fil. Elle fait en sorte que la méthode `run()` de l'objet soit invoquée dans un fil d'exécution.

Cette méthode lève une `RuntimeError` si elle est appelée plus d'une fois sur le même objet fil d'exécution.

If supported, set the operating system thread name to `threading.Thread.name`. The name can be truncated depending on the operating system thread name limits.

Modifié dans la version 3.14 : Set the operating system thread name.

run()

Méthode représentant l'activité du fil d'exécution.

Vous pouvez remplacer cette méthode dans une sous-classe. La méthode standard `run()` invoque l'objet appelable passé au constructeur de l'objet en tant qu'argument *target*, le cas échéant, avec des arguments positionnels et des arguments nommés tirés respectivement des arguments *args* et *kwargs*.

Using list or tuple as the *args* argument which passed to the `Thread` could achieve the same effect.

Example :

```
>>> from threading import Thread
>>> t = Thread(target=print, args=[1])
>>> t.run()
1
>>> t = Thread(target=print, args=(1,))
>>> t.run()
1
```

join(*timeout=None*)

Attend que le fil d'exécution se termine. Ceci bloque le fil appelant jusqu'à ce que le fil dont la méthode `join()` est appelée se termine – soit normalement, soit par une exception non gérée – ou jusqu'à ce que le délai optionnel *timeout* soit atteint.

When the *timeout* argument is present and not `None`, it should be a floating-point number specifying a timeout for the operation in seconds (or fractions thereof). As `join()` always returns `None`, you must call `is_alive()` after `join()` to decide whether a timeout happened -- if the thread is still alive, the `join()` call timed out.

Lorsque l'argument *timeout* n'est pas présent ou vaut `None`, l'opération se bloque jusqu'à ce que le fil d'exécution se termine.

A thread can be joined many times.

`join()` lève une `RuntimeError` si une tentative est faite pour attendre le fil d'exécution courant car cela conduirait à un interblocage (*deadlock* en anglais). Attendre via `join()` un fil d'exécution avant son lancement est aussi une erreur et, si vous tentez de le faire, lève la même exception.

name

Une chaîne de caractères utilisée à des fins d'identification seulement. Elle n'a pas de sémantique. Plusieurs fils d'exécution peuvent porter le même nom. Le nom initial est défini par le constructeur.

On some platforms, the thread name is set at the operating system level when the thread starts, so that it is visible in task managers. This name may be truncated to fit in a system-specific limit (for example, 15 bytes on Linux or 63 bytes on macOS).

Changes to *name* are only reflected at the OS level when the currently running thread is renamed. (Setting the *name* attribute of a different thread only updates the Python Thread object.)

getName()

setName()

Deprecated getter/setter API for `name`; use it directly as a property instead.

Obsolète depuis la version 3.10.

ident

« L'identificateur de fil d'exécution » de ce fil ou `None` si le fil n'a pas été lancé. C'est un entier non nul. Voyez également la fonction `get_ident()`. Les identificateurs de fils peuvent être recyclés lorsqu'un fil se termine et qu'un autre fil est créé. L'identifiant est disponible même après que le fil ait terminé.

native_id

The Thread ID (`TID`) of this thread, as assigned by the OS (kernel). This is a non-negative integer, or `None` if the thread has not been started. See the `get_native_id()` function. This value may be used to uniquely identify this particular thread system-wide (until the thread terminates, after which the value may be recycled by the OS).

> **Note**
>
> Tout comme pour les *Process IDs*, les *Thread IDs* ne sont valides (garantis uniques sur le système) uniquement du démarrage du fil à sa fin.

Availability : Windows, FreeBSD, Linux, macOS, OpenBSD, NetBSD, AIX, DragonFlyBSD.

Ajouté dans la version 3.8.

is_alive()

Renvoie si le fil d'exécution est vivant ou pas.

Cette méthode renvoie `True` depuis juste avant le démarrage de la méthode `run()` et jusqu'à juste après la terminaison de la méthode `run()`. La fonction `enumerate()` du module renvoie une liste de tous les fils d'exécution vivants.

daemon

A boolean value indicating whether this thread is a daemon thread (`True`) or not (`False`). This must be set before `start()` is called, otherwise `RuntimeError` is raised. Its initial value is inherited from the creating thread ; the main thread is not a daemon thread and therefore all threads created in the main thread default to `daemon` = `False`.

Le programme Python se termine lorsqu'il ne reste plus de fils d'exécution non-démons vivants.

isDaemon()

setDaemon()

Deprecated getter/setter API for `daemon` ; use it directly as a property instead.

Obsolète depuis la version 3.10.

18.1.3 Verrous

Un verrou primitif n'appartient pas à un fil d'exécution lorsqu'il est verrouillé. En Python, c'est actuellement la méthode de synchronisation la plus bas-niveau qui soit disponible, implémentée directement par le module d'extension `_thread`.

Un verrou primitif est soit « verrouillé » soit « déverrouillé ». Il est créé dans un état déverrouillé. Il a deux méthodes, `acquire()` et `release()`. Lorsque l'état est déverrouillé, `acquire()` verrouille et se termine immédiatement. Lorsque l'état est verrouillé, `acquire()` bloque jusqu'à ce qu'un appel à `release()` provenant d'un autre fil d'exécution le déverrouille. À ce moment `acquire()` le verrouille à nouveau et rend la main. La méthode `release()` ne doit être appelée que si le verrou est verrouillé, elle le déverrouille alors et se termine immédiatement. Déverrouiller un verrou qui n'est pas verrouillé provoque une `RuntimeError`.

Locks also support the *context management protocol*.

When more than one thread is blocked in `acquire()` waiting for the state to turn to unlocked, only one thread proceeds when a `release()` call resets the state to unlocked ; which one of the waiting threads proceeds is not defined, and may vary across implementations.

All methods are executed atomically.

class threading.**Lock**

The class implementing primitive lock objects. Once a thread has acquired a lock, subsequent attempts to acquire it block, until it is released ; any thread may release it.

Modifié dans la version 3.13 : `Lock` is now a class. In earlier Pythons, `Lock` was a factory function which returned an instance of the underlying private lock type.

acquire(*blocking=True*, *timeout=-1*)

Acquiert un verrou, bloquant ou non bloquant.

When invoked with the *blocking* argument set to `True` (the default), block until the lock is unlocked, then set it to locked and return `True`.

When invoked with the *blocking* argument set to `False`, do not block. If a call with *blocking* set to `True` would block, return `False` immediately ; otherwise, set the lock to locked and return `True`.

When invoked with the floating-point *timeout* argument set to a positive value, block for at most the number of seconds specified by *timeout* and as long as the lock cannot be acquired. A *timeout* argument of `-1` specifies an unbounded wait. It is forbidden to specify a *timeout* when *blocking* is `False`.

The return value is `True` if the lock is acquired successfully, `False` if not (for example if the *timeout* expired).

Modifié dans la version 3.2 : Le paramètre *timeout* est nouveau.

Modifié dans la version 3.2 : Lock acquisition can now be interrupted by signals on POSIX if the underlying threading implementation supports it.

Modifié dans la version 3.14 : Lock acquisition can now be interrupted by signals on Windows.

release()

Release a lock. This can be called from any thread, not only the thread which has acquired the lock.

When the lock is locked, reset it to unlocked, and return. If any other threads are blocked waiting for the lock to become unlocked, allow exactly one of them to proceed.

Lorsqu'elle est invoquée sur un verrou déverrouillé, une `RuntimeError` est levée.

Il n'y a pas de valeur de retour.

locked()

Return `True` if the lock is acquired.

18.1.4 RLock Objects

A reentrant lock is a synchronization primitive that may be acquired multiple times by the same thread. Internally, it uses the concepts of "owning thread" and "recursion level" in addition to the locked/unlocked state used by primitive locks. In the locked state, some thread owns the lock ; in the unlocked state, no thread owns it.

Threads call a lock's `acquire()` method to lock it, and its `release()` method to unlock it.

> **Note**
>
> Reentrant locks support the *context management protocol*, so it is recommended to use `with` instead of manually calling `acquire()` and `release()` to handle acquiring and releasing the lock for a block of code.

RLock's `acquire()`/`release()` call pairs may be nested, unlike Lock's `acquire()`/`release()`. Only the final `release()` (the `release()` of the outermost pair) resets the lock to an unlocked state and allows another thread blocked in `acquire()` to proceed.

`acquire()`/`release()` must be used in pairs : each acquire must have a release in the thread that has acquired the lock. Failing to call release as many times the lock has been acquired can lead to deadlock.

class threading.**RLock**

This class implements reentrant lock objects. A reentrant lock must be released by the thread that acquired it. Once a thread has acquired a reentrant lock, the same thread may acquire it again without blocking ; the thread must release it once for each time it has acquired it.

Note that `RLock` is actually a factory function which returns an instance of the most efficient version of the concrete RLock class that is supported by the platform.

acquire(*blocking=True*, *timeout=-1*)

Acquiert un verrou, bloquant ou non bloquant.

> **Voir aussi**
>
> ***Using RLock as a context manager***
> Recommended over manual `acquire()` and `release()` calls whenever practical.

When invoked with the *blocking* argument set to `True` (the default) :

— If no thread owns the lock, acquire the lock and return immediately.

— If another thread owns the lock, block until we are able to acquire lock, or *timeout*, if set to a positive float value.

— If the same thread owns the lock, acquire the lock again, and return immediately. This is the difference between `Lock` and `RLock` ; `Lock` handles this case the same as the previous, blocking until the lock can be acquired.

When invoked with the *blocking* argument set to `False` :

— If no thread owns the lock, acquire the lock and return immediately.

— If another thread owns the lock, return immediately.

— If the same thread owns the lock, acquire the lock again and return immediately.

In all cases, if the thread was able to acquire the lock, return `True`. If the thread was unable to acquire the lock (i.e. if not blocking or the timeout was reached) return `False`.

If called multiple times, failing to call `release()` as many times may lead to deadlock. Consider using `RLock` as a context manager rather than calling acquire/release directly.

Modifié dans la version 3.2 : Le paramètre *timeout* est nouveau.

release()

Release a lock, decrementing the recursion level. If after the decrement it is zero, reset the lock to unlocked (not owned by any thread), and if any other threads are blocked waiting for the lock to become unlocked, allow exactly one of them to proceed. If after the decrement the recursion level is still nonzero, the lock remains locked and owned by the calling thread.

Only call this method when the calling thread owns the lock. A `RuntimeError` is raised if this method is called when the lock is not acquired.

Il n'y a pas de valeur de retour.

18.1.5 Condition Objects

A condition variable is always associated with some kind of lock ; this can be passed in or one will be created by default. Passing one in is useful when several condition variables must share the same lock. The lock is part of the condition object : you don't have to track it separately.

A condition variable obeys the *context management protocol* : using the `with` statement acquires the associated lock for the duration of the enclosed block. The `acquire()` and `release()` methods also call the corresponding methods of the associated lock.

Other methods must be called with the associated lock held. The `wait()` method releases the lock, and then blocks until another thread awakens it by calling `notify()` or `notify_all()`. Once awakened, `wait()` re-acquires the lock and returns. It is also possible to specify a timeout.

The `notify()` method wakes up one of the threads waiting for the condition variable, if any are waiting. The `notify_all()` method wakes up all threads waiting for the condition variable.

Note : the `notify()` and `notify_all()` methods don't release the lock ; this means that the thread or threads awakened will not return from their `wait()` call immediately, but only when the thread that called `notify()` or `notify_all()` finally relinquishes ownership of the lock.

The typical programming style using condition variables uses the lock to synchronize access to some shared state ; threads that are interested in a particular change of state call `wait()` repeatedly until they see the desired state, while threads that modify the state call `notify()` or `notify_all()` when they change the state in such a way that it could possibly be a desired state for one of the waiters. For example, the following code is a generic producer-consumer situation with unlimited buffer capacity :

```
# Consume one item
with cv:
    while not an_item_is_available():
        cv.wait()
    get_an_available_item()

# Produce one item
with cv:
    make_an_item_available()
    cv.notify()
```

The `while` loop checking for the application's condition is necessary because `wait()` can return after an arbitrary long time, and the condition which prompted the `notify()` call may no longer hold true. This is inherent to multi-threaded programming. The `wait_for()` method can be used to automate the condition checking, and eases the computation of timeouts :

```
# Consume an item
with cv:
    cv.wait_for(an_item_is_available)
    get_an_available_item()
```

To choose between `notify()` and `notify_all()`, consider whether one state change can be interesting for only one or several waiting threads. E.g. in a typical producer-consumer situation, adding one item to the buffer only needs to wake up one consumer thread.

class threading.**Condition**(*lock=None*)

This class implements condition variable objects. A condition variable allows one or more threads to wait until they are notified by another thread.

If the *lock* argument is given and not `None`, it must be a `Lock` or `RLock` object, and it is used as the underlying lock. Otherwise, a new `RLock` object is created and used as the underlying lock.

Modifié dans la version 3.3 : changed from a factory function to a class.

acquire(**args*)

Acquire the underlying lock. This method calls the corresponding method on the underlying lock ; the return value is whatever that method returns.

release()

Release the underlying lock. This method calls the corresponding method on the underlying lock ; there is no return value.

wait(*timeout=None*)

Wait until notified or until a timeout occurs. If the calling thread has not acquired the lock when this method is called, a `RuntimeError` is raised.

This method releases the underlying lock, and then blocks until it is awakened by a `notify()` or `notify_all()` call for the same condition variable in another thread, or until the optional timeout occurs. Once awakened or timed out, it re-acquires the lock and returns.

When the *timeout* argument is present and not `None`, it should be a floating-point number specifying a timeout for the operation in seconds (or fractions thereof).

When the underlying lock is an `RLock`, it is not released using its `release()` method, since this may not actually unlock the lock when it was acquired multiple times recursively. Instead, an internal interface of the `RLock` class is used, which really unlocks it even when it has been recursively acquired several times. Another internal interface is then used to restore the recursion level when the lock is reacquired.

The return value is `True` unless a given *timeout* expired, in which case it is `False`.

Modifié dans la version 3.2 : Previously, the method always returned `None`.

wait_for(*predicate*, *timeout=None*)

Wait until a condition evaluates to true. *predicate* should be a callable which result will be interpreted as a boolean value. A *timeout* may be provided giving the maximum time to wait.

This utility method may call `wait()` repeatedly until the predicate is satisfied, or until a timeout occurs. The return value is the last return value of the predicate and will evaluate to `False` if the method timed out.

Ignoring the timeout feature, calling this method is roughly equivalent to writing :

```
while not predicate():
    cv.wait()
```

Therefore, the same rules apply as with `wait()` : The lock must be held when called and is re-acquired on return. The predicate is evaluated with the lock held.

Ajouté dans la version 3.2.

notify(*n=1*)

By default, wake up one thread waiting on this condition, if any. If the calling thread has not acquired the lock when this method is called, a `RuntimeError` is raised.

This method wakes up at most *n* of the threads waiting for the condition variable ; it is a no-op if no threads are waiting.

The current implementation wakes up exactly *n* threads, if at least *n* threads are waiting. However, it's not safe to rely on this behavior. A future, optimized implementation may occasionally wake up more than *n* threads.

Note : an awakened thread does not actually return from its `wait()` call until it can reacquire the lock. Since `notify()` does not release the lock, its caller should.

notify_all()

Wake up all threads waiting on this condition. This method acts like `notify()`, but wakes up all waiting threads instead of one. If the calling thread has not acquired the lock when this method is called, a `RuntimeError` is raised.

The method `notifyAll` is a deprecated alias for this method.

18.1.6 Semaphore Objects

This is one of the oldest synchronization primitives in the history of computer science, invented by the early Dutch computer scientist Edsger W. Dijkstra (he used the names `P()` and `V()` instead of `acquire()` and `release()`).

A semaphore manages an internal counter which is decremented by each `acquire()` call and incremented by each `release()` call. The counter can never go below zero ; when `acquire()` finds that it is zero, it blocks, waiting until some other thread calls `release()`.

Semaphores also support the *context management protocol*.

class threading.**Semaphore**(*value=1*)

This class implements semaphore objects. A semaphore manages an atomic counter representing the number of `release()` calls minus the number of `acquire()` calls, plus an initial value. The `acquire()` method blocks if necessary until it can return without making the counter negative. If not given, *value* defaults to 1.

The optional argument gives the initial *value* for the internal counter ; it defaults to `1`. If the *value* given is less than 0, `ValueError` is raised.

Modifié dans la version 3.3 : changed from a factory function to a class.

acquire(*blocking=True*, *timeout=None*)

Acquiert un sémaphore.

When invoked without arguments :

- If the internal counter is larger than zero on entry, decrement it by one and return `True` immediately.
- If the internal counter is zero on entry, block until awoken by a call to `release()`. Once awoken (and the counter is greater than 0), decrement the counter by 1 and return `True`. Exactly one thread will be awoken by each call to `release()`. The order in which threads are awoken should not be relied on.

When invoked with *blocking* set to `False`, do not block. If a call without an argument would block, return `False` immediately ; otherwise, do the same thing as when called without arguments, and return `True`.

When invoked with a *timeout* other than `None`, it will block for at most *timeout* seconds. If acquire does not complete successfully in that interval, return `False`. Return `True` otherwise.

Modifié dans la version 3.2 : Le paramètre *timeout* est nouveau.

release(*n=1*)

Release a semaphore, incrementing the internal counter by *n*. When it was zero on entry and other threads are waiting for it to become larger than zero again, wake up *n* of those threads.

Modifié dans la version 3.9 : Added the *n* parameter to release multiple waiting threads at once.

class threading.**BoundedSemaphore**(*value=1*)

Class implementing bounded semaphore objects. A bounded semaphore checks to make sure its current value doesn't exceed its initial value. If it does, `ValueError` is raised. In most situations semaphores are used to guard resources with limited capacity. If the semaphore is released too many times it's a sign of a bug. If not given, *value* defaults to 1.

Modifié dans la version 3.3 : changed from a factory function to a class.

`Semaphore` Example

Semaphores are often used to guard resources with limited capacity, for example, a database server. In any situation where the size of the resource is fixed, you should use a bounded semaphore. Before spawning any worker threads, your main thread would initialize the semaphore :

```
maxconnections = 5
# ...
pool_sema = BoundedSemaphore(value=maxconnections)
```

Once spawned, worker threads call the semaphore's acquire and release methods when they need to connect to the server :

```
with pool_sema:
    conn = connectdb()
    try:
        # ... use connection ...
    finally:
        conn.close()
```

The use of a bounded semaphore reduces the chance that a programming error which causes the semaphore to be released more than it's acquired will go undetected.

18.1.7 Event Objects

This is one of the simplest mechanisms for communication between threads : one thread signals an event and other threads wait for it.

An event object manages an internal flag that can be set to true with the `set()` method and reset to false with the `clear()` method. The `wait()` method blocks until the flag is true.

class threading.**Event**

Class implementing event objects. An event manages a flag that can be set to true with the `set()` method and reset to false with the `clear()` method. The `wait()` method blocks until the flag is true. The flag is initially false.

Modifié dans la version 3.3 : changed from a factory function to a class.

is_set()

Return `True` if and only if the internal flag is true.

The method `isSet` is a deprecated alias for this method.

set()

Set the internal flag to true. All threads waiting for it to become true are awakened. Threads that call `wait()` once the flag is true will not block at all.

clear()

Reset the internal flag to false. Subsequently, threads calling *wait()* will block until *set()* is called to set the internal flag to true again.

wait(*timeout=None*)

Block as long as the internal flag is false and the timeout, if given, has not expired. The return value represents the reason that this blocking method returned ; `True` if returning because the internal flag is set to true, or `False` if a timeout is given and the internal flag did not become true within the given wait time.

When the timeout argument is present and not `None`, it should be a floating-point number specifying a timeout for the operation in seconds, or fractions thereof.

Modifié dans la version 3.1 : Previously, the method always returned `None`.

18.1.8 Timer Objects

This class represents an action that should be run only after a certain amount of time has passed --- a timer. *Timer* is a subclass of *Thread* and as such also functions as an example of creating custom threads.

Timers are started, as with threads, by calling their *Timer.start* method. The timer can be stopped (before its action has begun) by calling the *cancel()* method. The interval the timer will wait before executing its action may not be exactly the same as the interval specified by the user.

Par exemple :

```
def hello():
    print("hello, world")

t = Timer(30.0, hello)
t.start()  # after 30 seconds, "hello, world" will be printed
```

class threading.**Timer**(*interval*, *function*, *args=None*, *kwargs=None*)

Create a timer that will run *function* with arguments *args* and keyword arguments *kwargs*, after *interval* seconds have passed. If *args* is `None` (the default) then an empty list will be used. If *kwargs* is `None` (the default) then an empty dict will be used.

Modifié dans la version 3.3 : changed from a factory function to a class.

cancel()

Stop the timer, and cancel the execution of the timer's action. This will only work if the timer is still in its waiting stage.

18.1.9 Barrier Objects

Ajouté dans la version 3.2.

This class provides a simple synchronization primitive for use by a fixed number of threads that need to wait for each other. Each of the threads tries to pass the barrier by calling the *wait()* method and will block until all of the threads have made their *wait()* calls. At this point, the threads are released simultaneously.

The barrier can be reused any number of times for the same number of threads.

As an example, here is a simple way to synchronize a client and server thread :

```
b = Barrier(2, timeout=5)

def server():
    start_server()
    b.wait()
    while True:
        connection = accept_connection()
```

(suite sur la page suivante)

(suite de la page précédente)

```
        process_server_connection(connection)

def client():
    b.wait()
    while True:
        connection = make_connection()
        process_client_connection(connection)
```

class threading.**Barrier**(*parties*, *action=None*, *timeout=None*)

Create a barrier object for *parties* number of threads. An *action*, when provided, is a callable to be called by one of the threads when they are released. *timeout* is the default timeout value if none is specified for the `wait()` method.

wait(*timeout=None*)

Pass the barrier. When all the threads party to the barrier have called this function, they are all released simultaneously. If a *timeout* is provided, it is used in preference to any that was supplied to the class constructor.

The return value is an integer in the range 0 to *parties* -- 1, different for each thread. This can be used to select a thread to do some special housekeeping, e.g. :

```
i = barrier.wait()
if i == 0:
    # Only one thread needs to print this
    print("passed the barrier")
```

If an *action* was provided to the constructor, one of the threads will have called it prior to being released. Should this call raise an error, the barrier is put into the broken state.

If the call times out, the barrier is put into the broken state.

This method may raise a `BrokenBarrierError` exception if the barrier is broken or reset while a thread is waiting.

reset()

Return the barrier to the default, empty state. Any threads waiting on it will receive the `BrokenBarrierError` exception.

Note that using this function may require some external synchronization if there are other threads whose state is unknown. If a barrier is broken it may be better to just leave it and create a new one.

abort()

Put the barrier into a broken state. This causes any active or future calls to `wait()` to fail with the `BrokenBarrierError`. Use this for example if one of the threads needs to abort, to avoid deadlocking the application.

It may be preferable to simply create the barrier with a sensible *timeout* value to automatically guard against one of the threads going awry.

parties

The number of threads required to pass the barrier.

n_waiting

The number of threads currently waiting in the barrier.

broken

Booléen qui vaut `True` si la barrière est rompue.

exception threading.**BrokenBarrierError**

Cette exception, une sous-classe de `RuntimeError`, est déclenchée lorsque l'objet `Barrier` est réinitialisé ou cassé.

18.1.10 Using locks, conditions, and semaphores in the `with` statement

All of the objects provided by this module that have `acquire` and `release` methods can be used as context managers for a `with` statement. The `acquire` method will be called when the block is entered, and `release` will be called when the block is exited. Hence, the following snippet :

```
with some_lock:
    # do something...
```

est équivalente à :

```
some_lock.acquire()
try:
    # do something...
finally:
    some_lock.release()
```

Currently, `Lock`, `RLock`, `Condition`, `Semaphore`, and `BoundedSemaphore` objects may be used as `with` statement context managers.

18.2 `multiprocessing` --- Process-based parallelism

Code source : Lib/multiprocessing/

Availability : not Android, not iOS, not WASI.

This module is not supported on *mobile platforms* or *WebAssembly platforms*.

18.2.1 Introduction

`multiprocessing` is a package that supports spawning processes using an API similar to the `threading` module. The `multiprocessing` package offers both local and remote concurrency, effectively side-stepping the *Global Interpreter Lock* by using subprocesses instead of threads. Due to this, the `multiprocessing` module allows the programmer to fully leverage multiple processors on a given machine. It runs on both POSIX and Windows.

Le module `multiprocessing` introduit aussi des API sans analogues dans le module `threading`. Un exemple est l'objet `Pool` qui offre un moyen simple de paralléliser l'exécution d'une fonction sur plusieurs valeurs d'entrée, en distribuant ces valeurs entre les processus (parallélisme de données). L'exemple suivant présente la manière classique de définir une telle fonction dans un module afin que les processus fils puissent importer ce module avec succès. Cet exemple basique de parallélisme de données, utilisant `Pool`,

```
from multiprocessing import Pool

def f(x):
    return x*x

if __name__ == '__main__':
    with Pool(5) as p:
        print(p.map(f, [1, 2, 3]))
```

affiche sur la sortie standard

```
[1, 4, 9]
```

Voir aussi

> `concurrent.futures.ProcessPoolExecutor` offers a higher level interface to push tasks to a background process without blocking execution of the calling process. Compared to using the `Pool` interface directly, the `concurrent.futures` API more readily allows the submission of work to the underlying process pool to be separated from waiting for the results.

La classe `Process`

Dans le module `multiprocessing`, les processus sont instanciés en créant un objet `Process` et en appelant sa méthode `start()`. La classe `Process` suit la même API que `threading.Thread`. Un exemple trivial d'un programme multi-processus est

```
from multiprocessing import Process

def f(name):
    print('hello', name)

if __name__ == '__main__':
    p = Process(target=f, args=('bob',))
    p.start()
    p.join()
```

Voici un exemple plus étoffé qui affiche les identifiants des processus créés :

```
from multiprocessing import Process
import os

def info(title):
    print(title)
    print('module name:', __name__)
    print('parent process:', os.getppid())
    print('process id:', os.getpid())

def f(name):
    info('function f')
    print('hello', name)

if __name__ == '__main__':
    info('main line')
    p = Process(target=f, args=('bob',))
    p.start()
    p.join()
```

La nécessité de la ligne `if __name__ == '__main__'` est expliquée dans la section *Lignes directrices de programmation*.

Contextes et méthodes de démarrage

Selon la plateforme, `multiprocessing` gère trois manières de démarrer un processus. Ces *méthodes de démarrage* sont

spawn
The parent process starts a fresh Python interpreter process. The child process will only inherit those resources necessary to run the process object's `run()` method. In particular, unnecessary file descriptors and handles from the parent process will not be inherited. Starting a process using this method is rather slow compared to using *fork* or *forkserver*.

Available on POSIX and Windows platforms. The default on Windows and macOS.

fork
Le processus parent utilise `os.fork()` pour *forker* l'interpréteur Python. Le processus fils, quand

il démarre, est effectivement identique au processus parent. Toutes les ressources du parent sont héritées par le fils. Notez qu'il est problématique de *forker* sans danger un processus *multi-threadé*.

Available on POSIX systems.

Modifié dans la version 3.14 : This is no longer the default start method on any platform. Code that requires *fork* must explicitly specify that via `get_context()` or `set_start_method()`.

Modifié dans la version 3.12 : If Python is able to detect that your process has multiple threads, the `os.fork()` function that this start method calls internally will raise a `DeprecationWarning`. Use a different start method. See the `os.fork()` documentation for further explanation.

forkserver

When the program starts and selects the *forkserver* start method, a server process is spawned. From then on, whenever a new process is needed, the parent process connects to the server and requests that it fork a new process. The fork server process is single threaded unless system libraries or preloaded imports spawn threads as a side-effect so it is generally safe for it to use `os.fork()`. No unnecessary resources are inherited.

Available on POSIX platforms which support passing file descriptors over Unix pipes such as Linux. The default on those.

Modifié dans la version 3.14 : This became the default start method on POSIX platforms.

Modifié dans la version 3.4 : *spawn* added on all POSIX platforms, and *forkserver* added for some POSIX platforms. Child processes no longer inherit all of the parents inheritable handles on Windows.

Modifié dans la version 3.8 : On macOS, the *spawn* start method is now the default. The *fork* start method should be considered unsafe as it can lead to crashes of the subprocess as macOS system libraries may start threads. See bpo-33725.

Modifié dans la version 3.14 : On POSIX platforms the default start method was changed from *fork* to *forkserver* to retain the performance but avoid common multithreaded process incompatibilities. See gh-84559.

On POSIX using the *spawn* or *forkserver* start methods will also start a *resource tracker* process which tracks the unlinked named system resources (such as named semaphores or `SharedMemory` objects) created by processes of the program. When all processes have exited the resource tracker unlinks any remaining tracked object. Usually there should be none, but if a process was killed by a signal there may be some "leaked" resources. (Neither leaked semaphores nor shared memory segments will be automatically unlinked until the next reboot. This is problematic for both objects because the system allows only a limited number of named semaphores, and shared memory segments occupy some space in the main memory.)

Pour sélectionner une méthode de démarrage, utilisez la fonction `set_start_method()` dans la clause `if __name__ == '__main__'` du module principal. Par exemple :

```
import multiprocessing as mp

def foo(q):
    q.put('hello')

if __name__ == '__main__':
    mp.set_start_method('spawn')
    q = mp.Queue()
    p = mp.Process(target=foo, args=(q,))
    p.start()
    print(q.get())
    p.join()
```

`set_start_method()` ne doit pas être utilisée plus d'une fois dans le programme.

Alternativement, vous pouvez utiliser `get_context()` pour obtenir un contexte. Les contextes ont la même API que le module *multiprocessing*, et permettent l'utilisation de plusieurs méthodes de démarrage dans un même programme.

```
import multiprocessing as mp

def foo(q):
    q.put('hello')

if __name__ == '__main__':
    ctx = mp.get_context('spawn')
    q = ctx.Queue()
    p = ctx.Process(target=foo, args=(q,))
    p.start()
    print(q.get())
    p.join()
```

Notez que les objets relatifs à un contexte ne sont pas forcément compatibles avec les processus d'un contexte différent. En particulier, les verrous créés avec le contexte *fork* ne peuvent pas être passés aux processus lancés avec les méthodes *spawn* ou *forkserver*.

Une bibliothèque qui veut utiliser une méthode de démarrage particulière devrait probablement faire appel à `get_context()` pour éviter d'interférer avec le choix de l'utilisateur de la bibliothèque.

Avertissement

The `'spawn'` and `'forkserver'` start methods generally cannot be used with "frozen" executables (i.e., binaries produced by packages like **PyInstaller** and **cx_Freeze**) on POSIX systems. The `'fork'` start method may work if code does not use threads.

Échange d'objets entre les processus

`multiprocessing` gère deux types de canaux de communication entre les processus :

Files (*queues*)

La classe `Queue` est un clone assez proche de `queue.Queue`. Par exemple :

```
from multiprocessing import Process, Queue

def f(q):
    q.put([42, None, 'hello'])

if __name__ == '__main__':
    q = Queue()
    p = Process(target=f, args=(q,))
    p.start()
    print(q.get())    # prints "[42, None, 'hello']"
    p.join()
```

Queues are thread and process safe. Any object put into a `multiprocessing` queue will be serialized.

Tubes (*pipes*)

La fonction `Pipe()` renvoie une paire d'objets de connexion connectés à un tube qui est par défaut duplex (à double sens). Par exemple :

```
from multiprocessing import Process, Pipe

def f(conn):
    conn.send([42, None, 'hello'])
    conn.close()

if __name__ == '__main__':
```

(suite sur la page suivante)

(suite de la page précédente)

```
    parent_conn, child_conn = Pipe()
    p = Process(target=f, args=(child_conn,))
    p.start()
    print(parent_conn.recv())   # prints "[42, None, 'hello']"
    p.join()
```

Les deux objets de connexion renvoyés par `Pipe()` représentent les deux extrémités d'un tube. Chaque objet de connexion possède (entre autres) des méthodes `send()` et `recv()`. Notez que les données d'un tube peuvent être corrompues si deux processus (ou fils d'exécution) essaient de lire ou d'écrire sur la même extrémité du tube en même temps. Bien évidemment, deux processus peuvent utiliser les deux extrémités différentes en même temps sans risque de corruption.

The `send()` method serializes the object and `recv()` re-creates the object.

Synchronisation entre processus

`multiprocessing` contient des équivalents à toutes les primitives de synchronisation de `threading`. Par exemple il est possible d'utiliser un verrou pour s'assurer qu'un seul processus à la fois écrit sur la sortie standard :

```
from multiprocessing import Process, Lock

def f(l, i):
    l.acquire()
    try:
        print('hello world', i)
    finally:
        l.release()

if __name__ == '__main__':
    lock = Lock()

    for num in range(10):
        Process(target=f, args=(lock, num)).start()
```

Sans le verrou, les sorties des différents processus risquent d'être mélangées.

Partager un état entre les processus

Comme mentionné plus haut, il est généralement préférable d'éviter autant que possible d'utiliser des états partagés en programmation concurrente. C'est particulièrement vrai quand plusieurs processus sont utilisés.

Cependant, si vous devez réellement partager des données, `multiprocessing` permet de le faire de deux manières.

Mémoire partagée

Les données peuvent être stockées dans une mémoire partagée en utilisant des `Value` ou des `Array`. Par exemple, le code suivant

```
from multiprocessing import Process, Value, Array

def f(n, a):
    n.value = 3.1415927
    for i in range(len(a)):
        a[i] = -a[i]

if __name__ == '__main__':
    num = Value('d', 0.0)
    arr = Array('i', range(10))
```

(suite sur la page suivante)

(suite de la page précédente)

```
    p = Process(target=f, args=(num, arr))
    p.start()
    p.join()

    print(num.value)
    print(arr[:])
```

affiche

```
3.1415927
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
```

Les arguments `'d'` et `'i'` utilisés à la création des `num` et `arr` sont des codes de types identiques à ceux du module *array* : `'d'` indique un flottant double-précision et `'i'` indique un entier signé. Ces objets peuvent être partagés sans problème entre processus ou fils d'exécution.

Pour plus de flexibilité dans l'utilisation de mémoire partagée, vous pouvez utiliser le module *multiprocessing.sharedctypes* qui permet la création d'objets arbitraires *ctypes* alloués depuis la mémoire partagée.

Processus serveur

Un objet gestionnaire renvoyé par *Manager()* contrôle un processus serveur qui détient les objets Python et autorise les autres processus à les manipuler à l'aide de mandataires.

Un gestionnaire renvoyé par *Manager()* prend en charge les types *list*, *dict*, *Namespace*, *Lock*, *RLock*, *Semaphore*, *BoundedSemaphore*, *Condition*, *Event*, *Barrier*, *Queue*, *Value* et *Array*. Par exemple,

```
from multiprocessing import Process, Manager

def f(d, l):
    d[1] = '1'
    d['2'] = 2
    d[0.25] = None
    l.reverse()

if __name__ == '__main__':
    with Manager() as manager:
        d = manager.dict()
        l = manager.list(range(10))

        p = Process(target=f, args=(d, l))
        p.start()
        p.join()

        print(d)
        print(l)
```

affiche

```
{0.25: None, 1: '1', '2': 2}
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

Les processus serveurs de gestionnaires sont plus flexibles que les mémoires partagées parce qu'ils peuvent gérer des types d'objets arbitraires. Aussi, un gestionnaire unique peut être partagé par les processus sur différentes machines à travers le réseau. Cependant, ils sont plus lents que les mémoires partagées.

Utiliser un pool de *workers*

La classe `Pool` représente un pool de processus de travail. Elle possède des méthodes qui permettent aux tâches d'être déchargées vers les processus de travail de différentes manières.

Par exemple :

```
from multiprocessing import Pool, TimeoutError
import time
import os

def f(x):
    return x*x

if __name__ == '__main__':
    # start 4 worker processes
    with Pool(processes=4) as pool:

        # print "[0, 1, 4,..., 81]"
        print(pool.map(f, range(10)))

        # print same numbers in arbitrary order
        for i in pool.imap_unordered(f, range(10)):
            print(i)

        # evaluate "f(20)" asynchronously
        res = pool.apply_async(f, (20,))      # runs in *only* one process
        print(res.get(timeout=1))             # prints "400"

        # evaluate "os.getpid()" asynchronously
        res = pool.apply_async(os.getpid, ()) # runs in *only* one process
        print(res.get(timeout=1))             # prints the PID of that process

        # launching multiple evaluations asynchronously *may* use more processes
        multiple_results = [pool.apply_async(os.getpid, ()) for i in range(4)]
        print([res.get(timeout=1) for res in multiple_results])

        # make a single worker sleep for 10 seconds
        res = pool.apply_async(time.sleep, (10,))
        try:
            print(res.get(timeout=1))
        except TimeoutError:
            print("We lacked patience and got a multiprocessing.TimeoutError")

        print("For the moment, the pool remains available for more work")

    # exiting the 'with'-block has stopped the pool
    print("Now the pool is closed and no longer available")
```

Notez que les méthodes d'un pool ne doivent être utilisées que par le processus qui l'a créée.

Note

Functionality within this package requires that the `__main__` module be importable by the children. This is covered in *Lignes directrices de programmation* however it is worth pointing out here. This means that some examples, such as the `multiprocessing.pool.Pool` examples will not work in the interactive interpreter. For example :

```
>>> from multiprocessing import Pool
```

```
>>> p = Pool(5)
>>> def f(x):
...     return x*x
...
>>> with p:
...     p.map(f, [1,2,3])
Process PoolWorker-1:
Process PoolWorker-2:
Process PoolWorker-3:
Traceback (most recent call last):
Traceback (most recent call last):
Traceback (most recent call last):
AttributeError: Can't get attribute 'f' on <module '__main__' (<class '_frozen_
↪importlib.BuiltinImporter'>)>
AttributeError: Can't get attribute 'f' on <module '__main__' (<class '_frozen_
↪importlib.BuiltinImporter'>)>
AttributeError: Can't get attribute 'f' on <module '__main__' (<class '_frozen_
↪importlib.BuiltinImporter'>)>
```

Si vous essayez ce code, il affichera trois traces d'appels complètes entrelacées de manière semi-aléatoire, et vous devrez vous débrouiller pour arrêter le processus maître.

18.2.2 Référence

Le paquet `multiprocessing` reproduit en grande partie l'API du module `threading`.

`Process` et exceptions

class multiprocessing.**Process**(*group=None*, *target=None*, *name=None*, *args=()*, *kwargs={}*, *, *daemon=None*)

Les objets *process* représentent une activité exécutée dans un processus séparé. La classe `Process` a des équivalents à toutes les méthodes de `threading.Thread`.

The constructor should always be called with keyword arguments. *group* should always be `None` ; it exists solely for compatibility with `threading.Thread`. *target* is the callable object to be invoked by the `run()` method. It defaults to `None`, meaning nothing is called. *name* is the process name (see `name` for more details). *args* is the argument tuple for the target invocation. *kwargs* is a dictionary of keyword arguments for the target invocation. If provided, the keyword-only *daemon* argument sets the process `daemon` flag to `True` or `False`. If `None` (the default), this flag will be inherited from the creating process.

By default, no arguments are passed to *target*. The *args* argument, which defaults to `()`, can be used to specify a list or tuple of the arguments to pass to *target*.

Si une sous-classe redéfinit le constructeur, elle doit s'assurer d'invoquer le constructeur de la classe de base (`Process.__init__()`) avant de faire autre chose du processus.

Modifié dans la version 3.3 : Added the *daemon* parameter.

run()

Méthode représentant l'activité du processus.

Vous pouvez redéfinir cette méthode dans une sous-classe. La méthode standard `run()` invoque l'objet appelable passé au constructeur comme argument *target*, si fourni, avec les arguments séquentiels et nommés respectivement pris depuis les paramètres *args* et *kwargs*.

Using a list or tuple as the *args* argument passed to `Process` achieves the same effect.

Example :

```
>>> from multiprocessing import Process
>>> p = Process(target=print, args=[1])
>>> p.run()
1
>>> p = Process(target=print, args=(1,))
>>> p.run()
1
```

start()

Démarre l'activité du processus.

Elle doit être appelée au plus une fois par objet processus. Elle s'arrange pour que la méthode `run()` de l'objet soit invoquée dans un processus séparé.

join([*timeout*])

Si l'argument optionnel *timeout* est `None` (par défaut), la méthode bloque jusqu'à ce que le processus dont la méthode `join()` a été appelée se termine. Si *timeout* est un nombre positif, elle bloque au maximum pendant *timeout* secondes. Notez que la méthode renvoie `None` si le processus se termine ou si le temps d'exécution expire. Vérifiez l'attribut `exitcode` du processus pour déterminer s'il s'est terminé.

join peut être appelée plusieurs fois sur un même processus.

Un processus ne peut pas s'attendre lui-même car cela causerait un interblocage. C'est une erreur d'essayer d'attendre un processus avant qu'il ne soit démarré.

name

Le nom du processus. Le nom est une chaîne de caractères utilisée uniquement pour l'identification du processus. Il n'a pas de sémantique. Plusieurs processus peuvent avoir le même nom.

Le nom initial est déterminé par le constructeur. Si aucun nom explicite n'est fourni au constructeur, un nom de la forme « Process-N_1 :N_2 :... :N_k » est construit, où chaque N_k est le N^e fils de son parent.

is_alive()

Renvoie vrai si le processus est en vie, faux sinon.

Grossièrement, un objet processus est en vie depuis le moment où la méthode `start()` finit de s'exécuter jusqu'à ce que le processus fils se termine.

daemon

L'option *daemon* du processus, une valeur booléenne. L'option doit être réglée avant que la méthode `start()` ne soit appelée.

La valeur initiale est héritée par le processus créateur.

Quand un processus se ferme, il tente de terminer tous ses processus fils *daemon*.

Notez qu'un processus *daemon* n'est pas autorisé à créer des processus fils. Sinon un processus *daemon* laisserait ses fils orphelins lorsqu'il se termine par la fermeture de son parent. De plus, ce **ne sont pas** des *daemons* ou services Unix, ce sont des processus normaux qui seront terminés (et non attendus) si un processus non *daemon* se ferme.

En plus de l'API `threading.Thread`, les objets `Process` supportent aussi les attributs et méthodes suivants :

pid

Renvoie l'ID du processus. Avant que le processus ne soit lancé, la valeur est `None`.

exitcode

The child's exit code. This will be `None` if the process has not yet terminated.

If the child's `run()` method returned normally, the exit code will be 0. If it terminated via `sys.exit()` with an integer argument *N*, the exit code will be *N*.

If the child terminated due to an exception not caught within `run()`, the exit code will be 1. If it was terminated by signal *N*, the exit code will be the negative value -*N*.

authkey

La clé d'authentification du processus (une chaîne d'octets).

Quand `multiprocessing` est initialisé, une chaîne aléatoire est assignée au processus principal, en utilisant `os.urandom()`.

Quand un objet `Process` est créé, il hérité de la clé d'authentification de son parent, bien que cela puisse être changé à l'aide du paramètre `authkey` pour une autre chaîne d'octets.

Voir *Clés d'authentification*.

sentinel

Un identifiant numérique de l'objet système qui devient « prêt » quand le processus se termine.

You can use this value if you want to wait on several events at once using `multiprocessing.connection.wait()`. Otherwise calling `join()` is simpler.

On Windows, this is an OS handle usable with the `WaitForSingleObject` and `WaitForMultipleObjects` family of API calls. On POSIX, this is a file descriptor usable with primitives from the `select` module.

Ajouté dans la version 3.3.

terminate()

Terminate the process. On POSIX this is done using the `SIGTERM` signal ; on Windows `TerminateProcess()` is used. Note that exit handlers and finally clauses, etc., will not be executed.

Notez que les descendants du processus ne sont *pas* terminés – ils deviendront simplement orphelins.

> **Avertissement**
>
> Si cette méthode est utilisée quand le processus associé utilise un tube ou une file, alors le tube ou la file sont susceptibles d'être corrompus et peuvent devenir inutilisables par les autres processus. De façon similaire, si le processus a acquis un verrou, un sémaphore ou autre, alors le terminer est susceptible de provoquer des blocages dans les autres processus.

kill()

Same as `terminate()` but using the `SIGKILL` signal on POSIX.

Ajouté dans la version 3.7.

close()

Ferme l'objet `Process`, libérant toutes les ressources qui lui sont associées. Une `ValueError` est levée si le processus sous-jacent tourne toujours. Une fois que `close()` se termine avec succès, la plupart des autres méthodes et attributs des objets `Process` lèveront une `ValueError`.

Ajouté dans la version 3.7.

Notez que les méthodes `start()`, `join()`, `is_alive()`, `terminate()` et `exitcode` ne doivent être appelées que par le processus ayant créé l'objet *process*.

Exemple d'utilisation de quelques méthodes de `Process` :

```
>>> import multiprocessing, time, signal
>>> mp_context = multiprocessing.get_context('spawn')
>>> p = mp_context.Process(target=time.sleep, args=(1000,))
>>> print(p, p.is_alive())
<...Process ... initial> False
>>> p.start()
>>> print(p, p.is_alive())
<...Process ... started> True
>>> p.terminate()
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> time.sleep(0.1)
>>> print(p, p.is_alive())
<...Process ... stopped exitcode=-SIGTERM> False
>>> p.exitcode == -signal.SIGTERM
True
```

exception multiprocessing.**ProcessError**

La classe de base de toutes les exceptions de *multiprocessing*.

exception multiprocessing.**BufferTooShort**

Exception raised by `Connection.recv_bytes_into()` when the supplied buffer object is too small for the message read.

Si `e` est une instance de *BufferTooShort* alors `e.args[0]` donnera un message sous forme d'une chaîne d'octets.

exception multiprocessing.**AuthenticationError**

Levée quand il y a une erreur d'authentification.

exception multiprocessing.**TimeoutError**

Levée par les méthodes avec temps d'exécution limité, quand ce temps expire.

Tubes (*pipes*) et files (*queues*)

Quand plusieurs processus travaillent ensemble, il est souvent nécessaire de les faire communiquer entre eux pour éviter d'avoir à utiliser des primitives de synchronisation comme les verrous.

Pour échanger des messages vous pouvez utiliser un *Pipe()* (pour une connexion entre deux processus) ou une file (qui autorise de plusieurs producteurs et consommateurs).

Les types *Queue*, *SimpleQueue* et *JoinableQueue* sont des files FIFO multi-producteurs et multi-consommateurs modelées sur la classe *queue.Queue* de la bibliothèque standard. Elles diffèrent par l'absence dans *Queue* des méthodes *task_done()* et *join()* introduites dans la classe *queue.Queue* par Python 2.5.

Si vous utilisez *JoinableQueue* alors vous **devez** appeler *JoinableQueue.task_done()* pour chaque tâche retirée de la file, sans quoi le sémaphore utilisé pour compter le nombre de tâches non accomplies pourra éventuellement déborder, levant une exception.

One difference from other Python queue implementations, is that *multiprocessing* queues serializes all objects that are put into them using *pickle*. The object return by the get method is a re-created object that does not share memory with the original object.

Notez que vous pouvez aussi créer une file partagée en utilisant un objet gestionnaire – voir *Gestionnaires*.

Note

multiprocessing utilise les exceptions habituelles *queue.Empty* et *queue.Full* pour signaler un dépassement du temps maximal autorisé. Elles ne sont pas disponibles dans l'espace de nommage *multiprocessing* donc vous devez les importer depuis le module *queue*.

Note

Quand un objet est placé dans une file, l'objet est sérialisé par *pickle* et un fil d'exécution en arrière-plan transmettra ensuite les données sérialisées sur un tube sous-jacent. Cela a certaines conséquences qui peuvent être un peu surprenantes, mais ne devrait causer aucune difficulté pratique — si elles vous embêtent vraiment, alors vous pouvez à la place utiliser une file créée avec un *manager*.

(1) Après avoir placé un objet dans une file vide il peut y avoir un délai infinitésimal avant que la méthode *empty()* de la file renvoie *False* et que *get_nowait()* renvoie une valeur sans lever de *queue.Empty*.

(2) Si plusieurs processus placent des objets dans la file, il est possible pour les objets d'être reçus de l'autre côté dans le désordre. Cependant, les objets placés par un même processus seront toujours récupérés dans l'ordre d'insertion.

Avertissement

Si un processus est tué à l'aide de `Process.terminate()` ou `os.kill()` pendant qu'il tente d'utiliser une `Queue`, alors les données de la file peuvent être corrompues. Cela peut par la suite causer des levées d'exceptions dans les autres processus quand ils tenteront d'utiliser la file.

Avertissement

Comme mentionné plus haut, si un processus fils a placé des éléments dans la file (et qu'il n'a pas utilisé `JoinableQueue.cancel_join_thread`), alors le processus ne se terminera pas tant que les éléments placés dans le tampon n'auront pas été transmis au tube.

Cela signifie que si vous essayez d'attendre ce processus vous pouvez obtenir un interblocage, à moins que vous ne soyez sûr que tous les éléments placés dans la file ont été consommés. De même, si le processus fils n'est pas un *daemon* alors le processus parent pourrait bloquer à la fermeture quand il tentera d'attendre tous ses fils non *daemons*.

Notez que la file créée à l'aide d'un gestionnaire n'a pas ce problème. Voir *Lignes directrices de programmation*.

Pour un exemple d'utilisation de files pour de la communication entre les processus, voir *Exemples*.

`multiprocessing.`**`Pipe`**`(`[*duplex*]`)`

Renvoie une paire `(conn1, conn2)` d'objets `Connection` représentant les extrémités d'un tube.

Si *duplex* vaut `True` (par défaut), alors le tube est bidirectionnel. Si *duplex* vaut `False` il est unidirectionnel : `conn1` ne peut être utilisé que pour recevoir des messages et `conn2` que pour en envoyer.

The `send()` method serializes the object using `pickle` and the `recv()` re-creates the object.

`class` `multiprocessing.`**`Queue`**`(`[*maxsize*]`)`

Renvoie une file partagée entre les processus utilisant un tube et quelques verrous / sémaphores. Quand un processus place initialement un élément sur la file, un fil d'exécution *chargeur* est démarré pour transférer les objets du tampon vers le tube.

Les exceptions habituelles `queue.Empty` et `queue.Full` du module `queue` de la bibliothèque standard sont levées pour signaler les *timeouts*.

`Queue` implémente toutes les méthodes de `queue.Queue` à l'exception de `task_done()` et `join()`.

`qsize``()`

Renvoie la taille approximative de la file. Ce nombre n'est pas fiable en raison des problématiques de *multithreading* et *multiprocessing*.

Note that this may raise `NotImplementedError` on platforms like macOS where `sem_getvalue()` is not implemented.

`empty``()`

Renvoie `True` si la file est vide, `False` sinon. Cette valeur n'est pas fiable en raison des problématiques de *multithreading* et *multiprocessing*.

May raise an `OSError` on closed queues. (not guaranteed)

`full``()`

Renvoie `True` si la file est pleine, `False` sinon. Cette valeur n'est pas fiable en raison des problématiques de *multithreading* et *multiprocessing*.

put(*obj*[, *block*[, *timeout*]])

Place *obj* dans la file. Si l'argument optionnel *block* vaut `True` (par défaut) est que *timeout* est `None` (par défaut), bloque jusqu'à ce qu'une place libre soit disponible. Si *timeout* est un nombre positif, la méthode bloquera au maximum *timeout* secondes et lève une exception `queue.Full` si aucune place libre n'a été trouvée dans le temps imparti. Autrement (*block* vaut `False`), place un élément dans la file si une place libre est immédiatement disponible, ou lève une exception `queue.Full` dans le cas contraire (*timeout* est ignoré dans ce cas).

Modifié dans la version 3.8 : si la file a été marquée comme fermée, une `ValueError` est levée. Auparavant, une `AssertionError` était levée.

put_nowait(*obj*)

Équivalent à `put(obj, False)`.

get([*block*[, *timeout*]])

Retire et renvoie un élément de la file. Si l'argument optionnel *block* vaut `True` (par défaut) et que *timeout* est `None` (par défaut), bloque jusqu'à ce qu'un élément soit disponible. Si *timeout* (le délai maximal autorisé) est un nombre positif, la méthode bloque au maximum *timeout* secondes et lève une exception `queue.Empty` si aucun élément n'est disponible dans le temps imparti. Autrement (*block* vaut `False`), renvoie un élément s'il est immédiatement disponible, ou lève une exception `queue.Empty` dans le cas contraire (*timeout* est ignoré dans ce cas).

Modifié dans la version 3.8 : si la file a été marquée comme terminée, une `ValueError` est levée. Auparavant, une `OSError`. était levée.

get_nowait()

Équivalent à `get(False)`.

`multiprocessing.Queue` possède quelques méthodes additionnelles non présentes dans `queue.Queue`. Ces méthodes ne sont habituellement pas nécessaires pour la plupart des codes :

close()

Indique que plus aucune donnée ne peut être placée sur la file par le processus courant. Le fil d'exécution en arrière-plan se terminera quand il aura transféré toutes les données du tampon vers le tube. Elle est appelée automatiquement quand la file est collectée par le ramasse-miettes.

join_thread()

Attend le fil d'exécution d'arrière-plan. Elle peut seulement être utilisée une fois que `close()` a été appelée. Elle bloque jusqu'à ce que le fil d'arrière-plan se termine, assurant que toutes les données du tampon ont été transmises au tube.

Par défaut si un processus n'est pas le créateur de la file alors à la fermeture elle essaie d'attendre le fil d'exécution d'arrière-plan de la file. Le processus peut appeler `cancel_join_thread()` pour que `join_thread()` ne fasse rien.

cancel_join_thread()

Empêche `join_thread()` de bloquer. En particulier, cela empêche le fil d'arrière-plan d'être attendu automatiquement quand le processus se ferme – voir `join_thread()`.

Un meilleur nom pour cette méthode pourrait être `allow_exit_without_flush()`. Cela peut provoquer des pertes de données placées dans la file, et il est très rare d'avoir besoin de l'utiliser. Elle n'est là que si vous souhaitez terminer immédiatement le processus sans transférer les données du tampon, et que vous êtes prêt à perdre des données.

Note

Le fonctionnement de cette classe requiert une implémentation de sémaphore partagé sur le système d'exploitation hôte. Sans cela, la fonctionnalité est désactivée et la tentative d'instancier une `Queue` lève une `ImportError`. Voir bpo-3770 pour plus d'informations. Cette remarque reste valable pour les autres types de files spécialisées définies par la suite.

class multiprocessing.**SimpleQueue**

Un type de *Queue* simplifié, très proche d'un *Pipe* avec verrou.

close()

Ferme la file : libère les ressources internes.

Une file ne doit plus être utilisée après sa fermeture. Par exemple, les méthodes *get()*, *put()* et *empty()* ne doivent plus être appelées.

Ajouté dans la version 3.9.

empty()

Renvoie `True` si la file est vide, `False` sinon.

Always raises an *OSError* if the SimpleQueue is closed.

get()

Supprime et renvoie un élément de la file.

put(*item*)

Place *item* dans la file.

class multiprocessing.**JoinableQueue**([*maxsize*])

JoinableQueue, une sous-classe de *Queue*, est une file qui ajoute des méthodes *task_done()* et *join()*.

task_done()

Indique qu'une tâche précédemment placée dans la file est achevée. Utilisée par les consommateurs de la file. Pour chaque *get()* utilisée pour récupérer une tâche, un appel ultérieur à *task_done()* indique à la file que le traitement de la tâche est terminé.

Si un *join()* est actuellement bloquant, il se débloquera quand tous les éléments auront été traités (signifiant qu'un appel à *task_done()* a été reçu pour chaque élément ayant été placé via *put()* dans la file).

Lève une exception *ValueError* si appelée plus de fois qu'il y avait d'éléments dans la file.

join()

Bloque jusqu'à ce que tous les éléments de la file aient été récupérés et traités.

Le compteur des tâches non accomplies augmente chaque fois qu'un élément est ajouté à la file. Le compteur redescend chaque fois qu'un consommateur appelle *task_done()* pour indiquer qu'un élément a été récupéré et que tout le travail qui le concerne est complété. Quand le compteur des tâches non accomplies atteint zéro, *join()* est débloquée.

Divers

multiprocessing.**active_children**()

Renvoie la liste de tous les fils vivants du processus courant.

Appeler cette méthode provoque l'effet de bord d'attendre tout processus qui n'a pas encore terminé.

multiprocessing.**cpu_count**()

Renvoie le nombre de CPUs sur le système.

This number is not equivalent to the number of CPUs the current process can use. The number of usable CPUs can be obtained with *os.process_cpu_count()* (or `len(os.sched_getaffinity(0))`).

Une *NotImplementedError* est levée quand il est impossible de déterminer ce nombre.

Voir aussi

os.cpu_count() *os.process_cpu_count()*

Modifié dans la version 3.13 : The return value can also be overridden using the `-X cpu_count` flag or `PYTHON_CPU_COUNT` as this is merely a wrapper around the `os` cpu count APIs.

`multiprocessing.current_process()`

Renvoie l'objet `Process` correspondant au processus courant.

Un analogue à `threading.current_thread()`.

`multiprocessing.parent_process()`

Renvoie l'objet `Process` correspondant au processus père de `current_process()`. Pour le processus maître, `parent_process` vaut `None`.

Ajouté dans la version 3.8.

`multiprocessing.freeze_support()`

Ajoute le support des programmes utilisant `multiprocessing` qui ont été figés pour produire un exécutable Windows (testé avec **py2exe**, **PyInstaller** et **cx_Freeze**).

Cette fonction doit être appelée juste après la ligne `if __name__ == '__main__'` du module principal. Par exemple :

```
from multiprocessing import Process, freeze_support

def f():
    print('hello world!')

if __name__ == '__main__':
    freeze_support()
    Process(target=f).start()
```

Si la ligne `freeze_support()` est omise, alors tenter de lancer l'exécutable figé lève une `RuntimeError`.

Appeler `freeze_support()` n'a pas d'effet quand elle est invoquée sur un système d'exploitation autre que Windows. De plus, si le module est lancé normalement par l'interpréteur Python sous Windows (le programme n'a pas été figé), alors `freeze_support()` n'a pas d'effet.

`multiprocessing.get_all_start_methods()`

Returns a list of the supported start methods, the first of which is the default. The possible start methods are `'fork'`, `'spawn'` and `'forkserver'`. Not all platforms support all methods. See *Contextes et méthodes de démarrage*.

Ajouté dans la version 3.4.

`multiprocessing.get_context(method=None)`

Renvoie un contexte ayant les mêmes attributs que le module `multiprocessing`.

If *method* is `None` then the default context is returned. Otherwise *method* should be `'fork'`, `'spawn'`, `'forkserver'`. `ValueError` is raised if the specified start method is not available. See *Contextes et méthodes de démarrage*.

Ajouté dans la version 3.4.

`multiprocessing.get_start_method(allow_none=False)`

Renvoie le nom de la méthode de démarrage utilisée pour démarrer le processus.

Si le nom de la méthode n'a pas été fixé et que *allow_none* est faux, alors la méthode de démarrage est réglée à celle par défaut et son nom est renvoyé. Si la méthode n'a pas été fixée et que *allow_none* est vrai, `None` est renvoyé.

The return value can be `'fork'`, `'spawn'`, `'forkserver'` or `None`. See *Contextes et méthodes de démarrage*.

Ajouté dans la version 3.4.

Modifié dans la version 3.8 : sur macOS, la méthode de démarrage *spawn* est maintenant la méthode par défaut. La méthode de démarrage *fork* doit être considérée comme dangereuse car elle peut entraîner des plantages du sous-processus. Voir bpo-33725.

`multiprocessing.`**`set_executable`**(*executable*)

Set the path of the Python interpreter to use when starting a child process. (By default `sys.executable` is used). Embedders will probably need to do some thing like

```
set_executable(os.path.join(sys.exec_prefix, 'pythonw.exe'))
```

avant de pouvoir créer des processus fils.

Modifié dans la version 3.4 : Now supported on POSIX when the `'spawn'` start method is used.

Modifié dans la version 3.11 : Accepte un *path-like object*.

`multiprocessing.`**`set_forkserver_preload`**(*module_names*)

Set a list of module names for the forkserver main process to attempt to import so that their already imported state is inherited by forked processes. Any `ImportError` when doing so is silently ignored. This can be used as a performance enhancement to avoid repeated work in every process.

For this to work, it must be called before the forkserver process has been launched (before creating a `Pool` or starting a `Process`).

Only meaningful when using the `'forkserver'` start method. See *Contextes et méthodes de démarrage*.

Ajouté dans la version 3.4.

`multiprocessing.`**`set_start_method`**(*method*, *force=False*)

Set the method which should be used to start child processes. The *method* argument can be `'fork'`, `'spawn'` or `'forkserver'`. Raises `RuntimeError` if the start method has already been set and *force* is not `True`. If *method* is `None` and *force* is `True` then the start method is set to `None`. If *method* is `None` and *force* is `False` then the context is set to the default context.

Notez que cette fonction ne devrait être appelée qu'une fois au plus, et l'appel devrait être protégé à l'intérieur d'une clause `if __name__ == '__main__'` dans le module principal.

See *Contextes et méthodes de démarrage*.

Ajouté dans la version 3.4.

Note

`multiprocessing` ne contient pas d'analogues à `threading.active_count()`, `threading.enumerate()`, `threading.settrace()`, `threading.setprofile()`, `threading.Timer`, ou `threading.local`.

Objets de connexions

Les objets de connexion autorisent l'envoi et la réception d'objets sérialisables ou de chaînes de caractères. Ils peuvent être vus comme des interfaces de connexion (*sockets*) connectées orientées messages.

Les objets de connexion sont habituellement créés via `Pipe` – voir aussi *Auditeurs et Clients*.

class `multiprocessing.connection.`**`Connection`**

`send`(*obj*)

Envoie un objet sur l'autre extrémité de la connexion, qui devra être lu avec `recv()`.

L'objet doit être sérialisable. Les *pickles* très larges (approximativement 32 Mo+, bien que cela dépende de l'OS) pourront lever une exception `ValueError`.

`recv`()

Renvoie un objet envoyé depuis l'autre extrémité de la connexion en utilisant `send()`. Bloque jusqu'à ce que quelque chose soit reçu. Lève une `EOFError` s'il n'y a plus rien à recevoir et que l'autre extrémité a été fermée.

fileno()

Renvoie le descripteur de fichier ou identifiant utilisé par la connexion.

close()

Ferme la connexion.

Elle est appelée automatiquement quand la connexion est collectée par le ramasse-miettes.

poll([*timeout*])

Renvoie vrai ou faux selon si des données sont disponibles à la lecture.

Si *timeout* n'est pas spécifié la méthode renverra immédiatement. Si *timeout* est un nombre alors il spécifie le temps maximum de blocage en secondes. Si *timeout* est `None`, un temps d'attente infini est utilisé.

Notez que plusieurs objets de connexions peuvent être attendus en même temps à l'aide de `multiprocessing.connection.wait()`.

send_bytes(*buffer*[, *offset*[, *size*]])

Envoie des données binaires depuis un *bytes-like object* comme un message complet.

Si *offset* est fourni, les données sont lues depuis cette position dans le tampon *buffer*. Si *size* est fourni, il indique le nombre d'octets qui seront lus depuis *buffer*. Les tampons très larges (approximativement 32 MiB+, bien que cela dépende de l'OS) pourront lever une exception `ValueError`.

recv_bytes([*maxlength*])

Renvoie un message complet de données binaires envoyées depuis l'autre extrémité de la connexion comme une chaîne de caractères. Bloque jusqu'à ce qu'il y ait quelque chose à recevoir. Lève une `EOFError` s'il ne reste rien à recevoir et que l'autre côté de la connexion a été fermé.

Si *maxlength* est précisé que que le message est plus long que *maxlength* alors une `OSError` est levée et la connexion n'est plus lisible.

Modifié dans la version 3.3 : Cette fonction levait auparavant une `IOError`, qui est maintenant un alias pour `OSError`.

recv_bytes_into(*buffer*[, *offset*])

Lit et stocke dans *buffer* un message complet de données binaires envoyées depuis l'autre extrémité de la connexion et renvoie le nombre d'octets du message. Bloque jusqu'à ce qu'il y ait quelque chose à recevoir. Lève une `EOFError` s'il ne reste rien à recevoir et que l'autre côté de la connexion a été fermé.

buffer doit être un *bytes-like object* accessible en écriture. Si *offset* est donné, le message sera écrit dans le tampon à partir de cette position. *offset* doit être un entier positif, inférieur à la taille de *buffer* (en octets).

Si le tampon est trop petit une exception `BufferTooShort` est levée et le message complet est accessible via `e.args[0]` où `e` est l'instance de l'exception.

Modifié dans la version 3.3 : Les objets de connexions eux-mêmes peuvent maintenant être transférés entre les processus en utilisant `Connection.send()` et `Connection.recv()`.

Connection objects also now support the context management protocol -- see *Le type gestionnaire de contexte*. `__enter__()` returns the connection object, and `__exit__()` calls `close()`.

Par exemple :

```
>>> from multiprocessing import Pipe
>>> a, b = Pipe()
>>> a.send([1, 'hello', None])
>>> b.recv()
[1, 'hello', None]
>>> b.send_bytes(b'thank you')
>>> a.recv_bytes()
b'thank you'
>>> import array
>>> arr1 = array.array('i', range(5))
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> arr2 = array.array('i', [0] * 10)
>>> a.send_bytes(arr1)
>>> count = b.recv_bytes_into(arr2)
>>> assert count == len(arr1) * arr1.itemsize
>>> arr2
array('i', [0, 1, 2, 3, 4, 0, 0, 0, 0, 0])
```

Avertissement

La méthode `Connection.recv()` désérialise automatiquement les données qu'elle reçoit, ce qui peut être un risque de sécurité à moins que vous ne fassiez réellement confiance au processus émetteur du message.

Par conséquent, à moins que l'objet de connexion soit instancié par `Pipe()`, vous ne devriez uniquement utiliser les méthodes `recv()` et `send()` après avoir effectué une quelconque forme d'authentification. Voir *Clés d'authentification*.

Avertissement

Si un processus est tué pendant qu'il essaye de lire ou écrire sur le tube, alors les données du tube ont des chances d'être corrompues, parce qu'il devient impossible d'être sûr d'où se trouvent les bornes du message.

Primitives de synchronisation

Généralement les primitives de synchronisation ne sont pas nécessaire dans un programme multi-processus comme elles le sont dans un programme multi-fils d'exécution. Voir la documentation du module `threading`.

Notez que vous pouvez aussi créer des primitives de synchronisation en utilisant un objet gestionnaire – voir *Gestionnaires*.

class `multiprocessing.`**`Barrier`**(*parties*[, *action*[, *timeout*]])

Un objet barrière : un clone de `threading.Barrier`.

Ajouté dans la version 3.3.

class `multiprocessing.`**`BoundedSemaphore`**([*value*])

Un objet sémaphore lié : un analogue proche de `threading.BoundedSemaphore`.

Une seule différence existe avec son proche analogue : le premier argument de sa méthode `acquire` est appelé *block*, pour la cohérence avec `Lock.acquire()`.

Note

Sur macOS, elle n'est pas distinguable de la classe `Semaphore` parce que `sem_getvalue()` n'est pas implémentée sur cette plateforme.

class `multiprocessing.`**`Condition`**([*lock*])

Une variable conditionnelle : un alias pour `threading.Condition`.

Si *lock* est spécifié il doit être un objet `Lock` ou `RLock` du module `multiprocessing`.

Modifié dans la version 3.3 : La méthode `wait_for()` a été ajoutée.

class `multiprocessing.`**`Event`**

Un clone de `threading.Event`.

class multiprocessing.**Lock**

Un verrou non récursif : un analogue proche de *threading.Lock*. Une fois que le processus ou le fil d'exécution a acquis un verrou, les tentatives suivantes d'acquisition depuis n'importe quel processus ou fil d'exécution bloqueront jusqu'à ce qu'il soit libéré ; n'importe quel processus ou fil peut le libérer. Les concepts et comportements de *threading.Lock* qui s'appliquent aux fils d'exécution sont répliqués ici dans *multiprocessing.Lock* et s'appliquent aux processus et aux fils d'exécution, à l'exception de ce qui est indiqué.

Notez que *Lock* est en fait une fonction *factory* qui renvoie une instance de multiprocessing.synchronize.Lock initialisée avec un contexte par défaut.

Lock supporte le protocole *context manager* et peut ainsi être utilisé avec une instruction with.

acquire(*block=True*, *timeout=None*)

Acquiert un verrou, bloquant ou non bloquant.

Avec l'argument *block* à True (par défaut), l'appel de méthode bloquera jusqu'à ce que le verrou soit dans déverrouillé, puis le verrouillera avant de renvoyer True. Notez que le nom de ce premier argument diffère de celui de *threading.Lock.acquire()*.

Avec l'argument *block* à False, l'appel de méthode ne bloque pas. Si le verrou est actuellement verrouillé, renvoie False ; autrement verrouille le verrou et renvoie True.

Quand invoqué avec un nombre flottant positif comme *timeout*, bloque au maximum pendant ce nombre spécifié de secondes, tant que le verrou ne peut être acquis. Les invocations avec une valeur de *timeout* négatives sont équivalents à zéro. Les invocations avec un *timeout* à None (par défaut) correspondent à un délai d'attente infini. Notez que le traitement des valeurs de *timeout* négatives et None diffère du comportement implémenté dans *threading.Lock.acquire()*. L'argument *timeout* n'a pas d'implication pratique si l'argument *block* est mis) False et est alors ignoré. Renvoie True si le verrou a été acquis et False si le temps de *timeout* a expiré.

release()

Libère un verrou. Elle peut être appelée depuis n'importe quel processus ou fil d'exécution, pas uniquement le processus ou le fil qui a acquis le verrou à l'origine.

Le comportement est le même que *threading.Lock.release()* excepté que lorsque la méthode est appelée sur un verrou déverrouillé, une *ValueError* est levée.

class multiprocessing.**RLock**

Un objet verrou récursif : un analogue proche de *threading.RLock*. Un verrou récursif doit être libéré par le processus ou le fil d'exécution qui l'a acquis. Quand un processus ou un fil acquiert un verrou récursif, le même processus/fil peut l'acquérir à nouveau sans bloquer ; le processus/fil doit le libérer autant de fois qu'il l'acquiert.

Notez que *RLock* est en fait une fonction *factory* qui renvoie une instance de multiprocessing.synchronize.RLock initialisée avec un contexte par défaut.

RLock supporte le protocole *context manager* et peut ainsi être utilisée avec une instruction with.

acquire(*block=True*, *timeout=None*)

Acquiert un verrou, bloquant ou non bloquant.

Quand invoqué avec l'argument *block* à True, bloque jusqu'à ce que le verrou soit déverrouillé (n'appartenant à aucun processus ou fil d'exécution) sauf s'il appartient déjà au processus ou fil d'exécution courant. Le processus ou fil d'exécution courant prend la possession du verrou (s'il ne l'a pas déjà) et incrémente d'un le niveau de récursion du verrou, renvoyant ainsi True. Notez qu'il y a plusieurs différences dans le comportement de ce premier argument comparé à l'implémentation de *threading.RLock.acquire()*, à commencer par le nom de l'argument lui-même.

Quand invoqué avec l'argument *block* à False, ne bloque pas. Si le verrou est déjà acquis (et possédé) par un autre processus ou fil d'exécution, le processus/fil courant n'en prend pas la possession et le niveau de récursion n'est pas incrémenté, résultant en une valeur de retour à False. Si le verrou est déverrouillé, le processus/fil courant en prend possession et incrémente son niveau de récursion, renvoyant True.

L'usage et les comportements de l'argument *timeout* sont les mêmes que pour *Lock.acquire()*. Notez que certains de ces comportements diffèrent par rapport à ceux implémentés par *threading.RLock.*

`acquire()`.

release()

Libère un verrou, décrémentant son niveau de récursion. Si après la décrémentation le niveau de récursion est zéro, réinitialise le verrou à un état déverrouillé (n'appartenant à aucun processus ou fil d'exécution) et si des processus/fils attendent que le verrou se déverrouillé, autorise un seul d'entre-eux à continuer. Si après cette décrémentation le niveau de récursion est toujours strictement positif, le verrou reste verrouillé et propriété du processus/fil appelant.

N'appelez cette méthode que si le processus ou fil d'exécution appelant est propriétaire du verrou. Une `AssertionError` est levée si cette méthode est appelée par un processus/fil autre que le propriétaire ou si le verrou n'est pas verrouillé (possédé). Notez que le type d'exception levé dans cette situation diffère du comportement de `threading.RLock.release()`.

class multiprocessing.**Semaphore**([*value*])

Un objet sémaphore, proche analogue de `threading.Semaphore`.

Une seule différence existe avec son proche analogue : le premier argument de sa méthode `acquire` est appelé *block*, pour la cohérence avec `Lock.acquire()`.

> **Note**
>
> Sous macOS, `sem_timedwait` n'est pas pris en charge, donc appeler `acquire()` avec un temps d'exécution limité émule le comportement de cette fonction en utilisant une boucle d'attente.

> **Note**
>
> Certaines des fonctionnalités de ce paquet requièrent une implémentation fonctionnelle de sémaphores partagés sur le système hôte. Sans cela, le module `multiprocessing.synchronize` sera désactivé, et les tentatives de l'importer lèveront une `ImportError`. Voir bpo-3770 pour plus d'informations.

Objets `ctypes` partagés

Il est possible de créer des objets partagés utilisant une mémoire partagée pouvant être héritée par les processus fils.

multiprocessing.**Value**(*typecode_or_type*, **args*, *lock=True*)

Renvoie un objet `ctypes` alloué depuis la mémoire partagée. Par défaut la valeur de retour est en fait un *wrapper* synchronisé autour de l'objet. L'objet en lui-même est accessible par l'attribut *value* de l'une `Value`.

typecode_or_type détermine le type de l'objet renvoyé : il s'agit soit d'un type *ctype* soit d'un caractère *typecode* tel qu'utilisé par le module `array`. **args* est passé au constructeur de ce type.

Si *lock* vaut `True` (par défaut), alors un nouveau verrou récursif est créé pour synchroniser l'accès à la valeur. Si *lock* est un objet `Lock` ou `RLock` alors il sera utilisé pour synchroniser l'accès à la valeur. Si *lock* vaut `False`, l'accès à l'objet renvoyé ne sera pas automatiquement protégé par un verrou, donc il ne sera pas forcément « *process-safe* ».

Les opérations telles que `+=` qui impliquent une lecture et une écriture ne sont pas atomiques. Ainsi si vous souhaitez par exemple réaliser une incrémentation atomique sur une valeur partagée, vous ne pouvez pas simplement faire

```
counter.value += 1
```

En supposant que le verrou associé est récursif (ce qui est le cas par défaut), vous pouvez à la place faire

```
with counter.get_lock():
    counter.value += 1
```

Notez que *lock* est un argument *keyword-only*.

`multiprocessing.`**`Array`**(*typecode_or_type*, *size_or_initializer*, *, *lock=True*)

Renvoie un tableau *ctypes* alloué depuis la mémoire partagée. Par défaut la valeur de retour est en fait un *wrapper* synchronisé autour du tableau.

typecode_or_type détermine le type des éléments du tableau renvoyé : il s'agit soit d'un type *ctype* soit d'un caractère *typecode* tel qu'utilisé par le module `array`. Si *size_or_initialize* est un entier, alors il détermine la taille du tableau, et le tableau sera initialisé avec des zéros. Autrement, *size*or_initializer* est une séquence qui sera utilisée pour initialiser le tableau et dont la taille détermine celle du tableau.

Si *lock* vaut `True` (par défaut), alors un nouveau verrou est créé pour synchroniser l'accès à la valeur. Si *lock* est un objet `Lock` ou `RLock` alors il sera utilisé pour synchroniser l'accès à la valeur. Si *lock* vaut `False`, l'accès à l'objet renvoyé ne sera pas automatiquement protégé par un verrou, donc il ne sera pas forcément « *process-safe* ».

Notez que *lock* est un argument *keyword-only*.

Notez qu'un tableau de `ctypes.c_char` a ses attributs *value* et *raw* qui permettent de l'utiliser pour stocker et récupérer des chaînes de caractères.

Le module `multiprocessing.sharedctypes`

Le module `multiprocessing.sharedctypes` fournit des fonctions pour allouer des objets `ctypes` depuis la mémoire partagée, qui peuvent être hérités par les processus fils.

> **Note**
>
> Bien qu'il soit possible de stocker un pointeur en mémoire partagée, rappelez-vous qu'un pointer référence un emplacement dans l'espace d'adressage d'un processus particulier. Ainsi, ce pointeur a de fortes chances d'être invalide dans le contexte d'un autre processus et déréférencer le pointeur depuis ce second processus peut causer un plantage.

`multiprocessing.sharedctypes.`**`RawArray`**(*typecode_or_type*, *size_or_initializer*)

Renvoie un tableau *ctypes* alloué depuis la mémoire partagée.

typecode_or_type détermine le type des éléments du tableau renvoyé : il s'agit soit d'un type *ctype* soit d'un caractère codant le type des éléments du tableau (*typecode*) tel qu'utilisé par le module `array`. Si *size_or_initialize* est un entier, alors il détermine la taille du tableau, et le tableau sera initialisé avec des zéros. Autrement, *size_or_initializer* est une séquence qui sera utilisée pour initialiser le tableau et dont la taille détermine celle du tableau.

Notez que définir ou récupérer un élément est potentiellement non atomique – utilisez plutôt `Array()` pour vous assurer de synchroniser automatiquement avec un verrou.

`multiprocessing.sharedctypes.`**`RawValue`**(*typecode_or_type*, **args*)

Renvoie un objet *ctypes* alloué depuis la mémoire partagée.

typecode_or_type détermine le type de l'objet renvoyé : il s'agit soit d'un type *ctype* soit d'un caractère *typecode* tel qu'utilisé par le module `array`. **args* est passé au constructeur de ce type.

Notez que définir ou récupérer un élément est potentiellement non atomique – utilisez plutôt `Value()` pour vous assurer de synchroniser automatiquement avec un verrou.

Notez qu'un tableau de `ctypes.c_char` a ses attributs *value* et *raw* qui permettent de l'utiliser pour stocker et récupérer des chaînes de caractères – voir la documentation de `ctypes`.

`multiprocessing.sharedctypes.`**`Array`**(*typecode_or_type*, *size_or_initializer*, *, *lock=True*)

Identique à `RawArray()` à l'exception que suivant la valeur de *lock* un *wrapper* de synchronisation *process-safe* pourra être renvoyé à la place d'un tableau *ctypes* brut.

Si *lock* vaut `True` (par défaut), alors un nouveau verrou est créé pour synchroniser l'accès à la valeur. Si *lock* est un objet `Lock` ou `RLock` alors il sera utilisé pour synchroniser l'accès à la valeur. Si *lock* vaut `False`,

l'accès à l'objet renvoyé ne sera pas automatiquement protégé par un verrou, donc il ne sera pas forcément « *process-safe* ».

Notez que *lock* est un argument *keyword-only*.

`multiprocessing.sharedctypes.`**`Value`**(*typecode_or_type*, **args*, *lock=True*)

Identique à `RawValue()` à l'exception que suivant la valeur de *lock* un *wrapper* de synchronisation *process-safe* pourra être renvoyé à la place d'un objet *ctypes* brut.

Si *lock* vaut `True` (par défaut), alors un nouveau verrou est créé pour synchroniser l'accès à la valeur. Si *lock* est un objet `Lock` ou `RLock` alors il sera utilisé pour synchroniser l'accès à la valeur. Si *lock* vaut `False`, l'accès à l'objet renvoyé ne sera pas automatiquement protégé par un verrou, donc il ne sera pas forcément « *process-safe* ».

Notez que *lock* est un argument *keyword-only*.

`multiprocessing.sharedctypes.`**`copy`**(*obj*)

Renvoie un objet *ctypes* alloué depuis la mémoire partagée, qui est une copie de l'objet *ctypes obj*.

`multiprocessing.sharedctypes.`**`synchronized`**(*obj*[, *lock*])

Renvoie un *wrapper process-safe* autour de l'objet *ctypes* qui utilise *lock* pour synchroniser l'accès. Si *lock* est `None` (par défaut), un objet `multiprocessing.RLock` est créé automatiquement.

Un *wrapper* synchronisé aura deux méthodes en plus de celles de l'objet qu'il enveloppe : `get_obj()` renvoie l'objet *wrappé* et `get_lock()` renvoie le verrou utilisé pour la synchronisation.

Notez qu'accéder à l'objet *ctypes* à travers le *wrapper* peut s'avérer beaucoup plus lent qu'accéder directement à l'objet *ctypes* brut.

Modifié dans la version 3.5 : Les objets synchronisés supportent le protocole *context manager*.

Le tableau ci-dessous compare la syntaxe de création des objets *ctypes* partagés depuis une mémoire partagée avec la syntaxe normale *ctypes*. (Dans le tableau, `MyStruct` est une sous-classe quelconque de `ctypes.Structure`.)

ctypes	*sharedctypes* utilisant un type	*sharedctypes* utilisant un *typecode*
c_double(2.4)	RawValue(c_double, 2.4)	RawValue('d', 2.4)
MyStruct(4, 6)	RawValue(MyStruct, 4, 6)	
(c_short * 7)()	RawArray(c_short, 7)	RawArray('h', 7)
(c_int * 3)(9, 2, 8)	RawArray(c_int, (9, 2, 8))	RawArray('i', (9, 2, 8))

Ci-dessous un exemple où des objets *ctypes* sont modifiés par un processus fils :

```
from multiprocessing import Process, Lock
from multiprocessing.sharedctypes import Value, Array
from ctypes import Structure, c_double

class Point(Structure):
    _fields_ = [('x', c_double), ('y', c_double)]

def modify(n, x, s, A):
    n.value **= 2
    x.value **= 2
    s.value = s.value.upper()
    for a in A:
        a.x **= 2
        a.y **= 2

if __name__ == '__main__':
    lock = Lock()
```

(suite sur la page suivante)

(suite de la page précédente)

```
    n = Value('i', 7)
    x = Value(c_double, 1.0/3.0, lock=False)
    s = Array('c', b'hello world', lock=lock)
    A = Array(Point, [(1.875,-6.25), (-5.75,2.0), (2.375,9.5)], lock=lock)

    p = Process(target=modify, args=(n, x, s, A))
    p.start()
    p.join()

    print(n.value)
    print(x.value)
    print(s.value)
    print([(a.x, a.y) for a in A])
```

Les résultats affichés sont

```
49
0.1111111111111111
HELLO WORLD
[(3.515625, 39.0625), (33.0625, 4.0), (5.640625, 90.25)]
```

Gestionnaires

Les gestionnaires fournissent un moyen de créer des données qui peuvent être partagées entre les différents processus, incluant le partage à travers le réseau entre des processus tournant sur des machines différentes. Un objet gestionnaire contrôle un processus serveur qui gère les *shared objects*. Les autres processus peuvent accéder aux objets partagés à l'aide de mandataires.

`multiprocessing.Manager()`

Renvoie un objet `SyncManager` démarré qui peut être utilisé pour partager des objets entre les processus. L'objet gestionnaire renvoyé correspond à un processus fils instancié et possède des méthodes pour créer des objets partagés et renvoyer les mandataires correspondants.

Les processus gestionnaires seront arrêtés dès qu'ils seront collectés par le ramasse-miettes ou que leur processus parent se terminera. Les classes gestionnaires sont définies dans le module `multiprocessing.managers` :

class `multiprocessing.managers.BaseManager`(*address=None*, *authkey=None*, *serializer='pickle'*, *ctx=None*, **, shutdown_timeout=1.0*)

Crée un objet *BaseManager*.

Une fois créé il faut appeler `start()` ou `get_server().serve_forever()` pour assurer que l'objet gestionnaire référence un processus gestionnaire démarré.

address est l'adresse sur laquelle le processus gestionnaire écoute pour de nouvelles connexions. Si *address* est `None`, une adresse arbitraire est choisie.

authkey est la clé d'authentification qui sera utilisée pour vérifier la validité des connexions entrantes sur le processus serveur. Si *authkey* est `None` alors `current_process().authkey` est utilisée. Autrement *authkey* est utilisée et doit être une chaîne d'octets.

serializer must be `'pickle'` (use `pickle` serialization) or `'xmlrpclib'` (use `xmlrpc.client` serialization).

ctx is a context object, or `None` (use the current context). See the `get_context()` function.

shutdown_timeout is a timeout in seconds used to wait until the process used by the manager completes in the `shutdown()` method. If the shutdown times out, the process is terminated. If terminating the process also times out, the process is killed.

Modifié dans la version 3.11 : Added the *shutdown_timeout* parameter.

start([*initializer*[, *initargs*]])

Démarre un sous-processus pour démarrer le gestionnaire. Si *initializer* n'est pas `None` alors le sous-processus appellera `initializer(*initargs)` quand il démarrera.

get_server()

Renvoie un objet `Server` qui représente le serveur sous le contrôle du gestionnaire. L'objet `Server` supporte la méthode `serve_forever()` :

```
>>> from multiprocessing.managers import BaseManager
>>> manager = BaseManager(address=('', 50000), authkey=b'abc')
>>> server = manager.get_server()
>>> server.serve_forever()
```

`Server` possède en plus un attribut *address*.

connect()

Connecte un objet gestionnaire local au processus gestionnaire distant :

```
>>> from multiprocessing.managers import BaseManager
>>> m = BaseManager(address=('127.0.0.1', 50000), authkey=b'abc')
>>> m.connect()
```

shutdown()

Stoppe le processus utilisé par le gestionnaire. Cela est disponible uniquement si *start()* a été utilisée pour démarrer le processus serveur.

Cette méthode peut être appelée plusieurs fois.

register(*typeid*[, *callable*[, *proxytype*[, *exposed*[, *method_to_typeid*[, *create_method*]]]]])

Une méthode de classe qui peut être utilisée pour enregistrer un type ou un appelable avec la classe gestionnaire.

typeif est un « *type identifier* » qui est utilisé pour identifier un type particulier d'objet partagé. Cela doit être une chaîne de caractères.

callable est un objet appelable utilisé pour créer les objets avec cet identifiant de type. Si une instance de gestionnaire prévoit de se connecter au serveur en utilisant sa méthode *connect()* ou si l'argument *create_method* vaut `False` alors cet argument peut être laissé à `None`.

proxytype est une sous-classe de *BaseProxy* utilisée pour créer des mandataires autour des objets partagés avec ce *typeid*. S'il est `None`, une classe mandataire sera créée automatiquement.

exposed est utilisé pour préciser une séquence de noms de méthodes dont les mandataires pour ce *typeid* doivent être autorisés à accéder via *BaseProxy._callmethod()*. (Si *exposed* est `None` alors `proxytype._exposed_` est utilisé à la place s'il existe.) Dans le cas où aucune liste *exposed* n'est précisée, toutes les « méthodes publiques » de l'objet partagé seront accessibles. (Ici une « méthode publique » signifie n'importe quel attribut qui possède une méthode `__call__()` et dont le nom ne commence pas par un `'_'`.)

method_to_typeid est un tableau associatif utilisé pour préciser le type de retour de ces méthodes exposées qui doivent renvoyer un mandataire. Il associé un nom de méthode à une chaîne de caractères *typeid*. (Si *method_to_typeid* est `None`, `proxytype._method_to_typeid_` est utilisé à la place s'il existe). Si le nom d'une méthode n'est pas une clé de ce tableau associatif ou si la valeur associée est `None`, l'objet renvoyé par la méthode sera une copie de la valeur.

create_method détermine si une méthode devrait être créée avec le nom *typeid*, qui peut être utilisée pour indiquer au processus serveur de créer un nouvel objet partagé et d'en renvoyer un mandataire. a valeur par défaut est `True`.

Les instances de *BaseManager* ont aussi une propriété en lecture seule :

address

L'adresse utilisée par le gestionnaire.

Modifié dans la version 3.3 : Les objets gestionnaires supportent le protocole des gestionnaires de contexte – voir *Le type gestionnaire de contexte*. `__enter__()` démarre le processus serveur (s'il n'a pas déjà été démarré) et renvoie l'objet gestionnaire. `__exit__()` appelle `shutdown()`.

Dans les versions précédentes `__enter__()` ne démarrait pas le processus serveur du gestionnaire s'il n'était pas déjà démarré.

class `multiprocessing.managers.`**`SyncManager`**

Une sous-classe de `BaseManager` qui peut être utilisée pour la synchronisation entre processus. Des objets de ce type sont renvoyés par `multiprocessing.Manager()`.

Ces méthodes créent et renvoient des *Objets mandataires* pour un certain nombre de types de données communément utilisés pour être synchronisés entre les processus. Elles incluent notamment des listes et dictionnaires partagés.

`Barrier`(*parties*[, *action*[, *timeout*]])

Crée un objet `threading.Barrier` partagé et renvoie un mandataire pour cet objet.

Ajouté dans la version 3.3.

`BoundedSemaphore`([*value*])

Crée un objet `threading.BoundedSemaphore` partagé et renvoie un mandataire pour cet objet.

`Condition`([*lock*])

Crée un objet `threading.Condition` partagé et renvoie un mandataire pour cet objet.

Si *lock* est fourni alors il doit être un mandataire pour un objet `threading.Lock` ou `threading.RLock`.

Modifié dans la version 3.3 : La méthode `wait_for()` a été ajoutée.

`Event`()

Crée un objet `threading.Event` partagé et renvoie un mandataire pour cet objet.

`Lock`()

Crée un objet `threading.Lock` partagé et renvoie un mandataire pour cet objet.

`Namespace`()

Crée un objet `Namespace` partagé et renvoie un mandataire pour cet objet.

`Queue`([*maxsize*])

Crée un objet `queue.Queue` partagé et renvoie un mandataire pour cet objet.

`RLock`()

Crée un objet `threading.RLock` partagé et renvoie un mandataire pour cet objet.

`Semaphore`([*value*])

Crée un objet `threading.Semaphore` partagé et renvoie un mandataire pour cet objet.

`Array`(*typecode*, *sequence*)

Crée un tableau et renvoie un mandataire pour cet objet.

`Value`(*typecode*, *value*)

Crée un objet avec un attribut `value` accessible en écriture et renvoie un mandataire pour cet objet.

`dict`()

`dict`(*mapping*)

`dict`(*sequence*)

Crée un objet `dict` partagé et renvoie un mandataire pour cet objet.

`list`()

`list`(*sequence*)

Crée un objet `list` partagé et renvoie un mandataire pour cet objet.

Modifié dans la version 3.6 : Les objets partagés peuvent être imbriqués. Par exemple, un conteneur partagé tel qu'une liste partagée peu contenir d'autres objets partagés qui seront aussi gérés et synchronisés par le *SyncManager*.

class multiprocessing.managers.**Namespace**

Un type qui peut être enregistré avec *SyncManager*.

Un espace de nommage n'a pas de méthodes publiques, mais possède des attributs accessibles en écriture. Sa représentation montre les valeurs de ses attributs.

Cependant, en utilisant un mandataire pour un espace de nommage, un attribut débutant par `'_'` est un attribut du mandataire et non de l'objet cible :

```
>>> mp_context = multiprocessing.get_context('spawn')
>>> manager = mp_context.Manager()
>>> Global = manager.Namespace()
>>> Global.x = 10
>>> Global.y = 'hello'
>>> Global._z = 12.3    # this is an attribute of the proxy
>>> print(Global)
Namespace(x=10, y='hello')
```

Gestionnaires personnalisés

Pour créer son propre gestionnaire, il faut créer une sous-classe de *BaseManager* et utiliser la méthode de classe *register()* pour enregistrer de nouveaux types ou *callables* au gestionnaire. Par exemple :

```
from multiprocessing.managers import BaseManager

class MathsClass:
    def add(self, x, y):
        return x + y
    def mul(self, x, y):
        return x * y

class MyManager(BaseManager):
    pass

MyManager.register('Maths', MathsClass)

if __name__ == '__main__':
    with MyManager() as manager:
        maths = manager.Maths()
        print(maths.add(4, 3))         # prints 7
        print(maths.mul(7, 8))         # prints 56
```

Utiliser un gestionnaire distant

Il est possible de lancer un serveur gestionnaire sur une machine et d'avoir des clients l'utilisant sur d'autres machines (en supposant que les pare-feus impliqués l'autorisent).

Exécuter les commandes suivantes crée un serveur pour une file simple partagée à laquelle des clients distants peuvent accéder :

```
>>> from multiprocessing.managers import BaseManager
>>> from queue import Queue
>>> queue = Queue()
>>> class QueueManager(BaseManager): pass
>>> QueueManager.register('get_queue', callable=lambda:queue)
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> m = QueueManager(address=('', 50000), authkey=b'abracadabra')
>>> s = m.get_server()
>>> s.serve_forever()
```

Un client peut accéder au serveur comme suit :

```
>>> from multiprocessing.managers import BaseManager
>>> class QueueManager(BaseManager): pass
>>> QueueManager.register('get_queue')
>>> m = QueueManager(address=('foo.bar.org', 50000), authkey=b'abracadabra')
>>> m.connect()
>>> queue = m.get_queue()
>>> queue.put('hello')
```

Un autre client peut aussi l'utiliser :

```
>>> from multiprocessing.managers import BaseManager
>>> class QueueManager(BaseManager): pass
>>> QueueManager.register('get_queue')
>>> m = QueueManager(address=('foo.bar.org', 50000), authkey=b'abracadabra')
>>> m.connect()
>>> queue = m.get_queue()
>>> queue.get()
'hello'
```

Les processus locaux peuvent aussi accéder à cette file, utilisant le code précédent sur le client pour y accéder à distance :

```
>>> from multiprocessing import Process, Queue
>>> from multiprocessing.managers import BaseManager
>>> class Worker(Process):
...     def __init__(self, q):
...         self.q = q
...         super().__init__()
...     def run(self):
...         self.q.put('local hello')
...
>>> queue = Queue()
>>> w = Worker(queue)
>>> w.start()
>>> class QueueManager(BaseManager): pass
...
>>> QueueManager.register('get_queue', callable=lambda: queue)
>>> m = QueueManager(address=('', 50000), authkey=b'abracadabra')
>>> s = m.get_server()
>>> s.serve_forever()
```

Objets mandataires

Un mandataire est un objet qui *référence* un objet partagé appartenant (supposément) à un processus différent. L'objet partagé est appelé le *référent* du mandataire. Plusieurs mandataires peuvent avoir un même référent.

Un mandataire possède des méthodes qui appellent les méthodes correspondantes du référent (bien que toutes les méthodes du référent ne soient pas nécessairement accessibles à travers le mandataire). De cette manière, un mandataire peut être utilisé comme le serait sont référent :

```
>>> mp_context = multiprocessing.get_context('spawn')
>>> manager = mp_context.Manager()
>>> l = manager.list([i*i for i in range(10)])
>>> print(l)
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> print(repr(l))
<ListProxy object, typeid 'list' at 0x...>
>>> l[4]
16
>>> l[2:5]
[4, 9, 16]
```

Notez qu'appliquer `str()` à un mandataire renvoie la représentation du référent, alors que `repr()` renvoie celle du mandataire.

Une fonctionnalité importantes des objets mandataires est qu'ils sont sérialisables et peuvent donc être échangés entre les processus. Ainsi, un référent peut contenir des *Objets mandataires*. Cela permet d'imbriquer des listes et dictionnaires gérés ainsi que d'autres *Objets mandataires* :

```
>>> a = manager.list()
>>> b = manager.list()
>>> a.append(b)         # referent of a now contains referent of b
>>> print(a, b)
[<ListProxy object, typeid 'list' at ...>] []
>>> b.append('hello')
>>> print(a[0], b)
['hello'] ['hello']
```

De même, les mandataires de listes et dictionnaires peuvent être imbriqués dans d'autres :

```
>>> l_outer = manager.list([ manager.dict() for i in range(2) ])
>>> d_first_inner = l_outer[0]
>>> d_first_inner['a'] = 1
>>> d_first_inner['b'] = 2
>>> l_outer[1]['c'] = 3
>>> l_outer[1]['z'] = 26
>>> print(l_outer[0])
{'a': 1, 'b': 2}
>>> print(l_outer[1])
{'c': 3, 'z': 26}
```

Si des objets standards (non *proxyfiés*) `list` ou `dict` sont contenus dans un référent, les modifications sur ces valeurs mutables ne seront pas propagées à travers le gestionnaire parce que le mandataire n'a aucun moyen de savoir quand les valeurs contenues sont modifiées. Cependant, stocker une valeur dans un conteneur mandataire (qui déclenche un appel à `__setitem__` sur le mandataire) propage bien la modification à travers le gestionnaire et modifie effectivement l'élément, il est ainsi possible de réassigner la valeur modifiée au conteneur mandataire :

```
# create a list proxy and append a mutable object (a dictionary)
lproxy = manager.list()
lproxy.append({})
# now mutate the dictionary
d = lproxy[0]
d['a'] = 1
d['b'] = 2
# at this point, the changes to d are not yet synced, but by
# updating the dictionary, the proxy is notified of the change
lproxy[0] = d
```

Cette approche est peut-être moins pratique que d'utiliser des *Objets mandataires* imbriqués pour la majorité des cas

d'utilisation, mais démontre aussi un certain niveau de contrôle sur la synchronisation.

Note

Les types de mandataires de *multiprocessing* n'implémentent rien pour la comparaison par valeurs. Par exemple, on a :

```
>>> manager.list([1,2,3]) == [1,2,3]
False
```

Il faut à la place simplement utiliser une copie du référent pour faire les comparaisons.

class multiprocessing.managers.**BaseProxy**

Les objets mandataires sont des instances de sous-classes de *BaseProxy*.

_callmethod(*methodname*[, *args*[, *kwds*]])

Appelle et renvoie le résultat d'une méthode du référent du mandataire.

Si `proxy` est un mandataire sont le référent est `obj`, alors l'expression

```
proxy._callmethod(methodname, args, kwds)
```

s'évalue comme

```
getattr(obj, methodname)(*args, **kwds)
```

dans le processus du gestionnaire.

La valeur renvoyée sera une copie du résultat de l'appel ou un mandataire sur un nouvel objet partagé – voir l'a documentation de l'argument *method_to_typeid* de *BaseManager.register()*.

Si une exception est levée par l'appel, elle est relayée par *_callmethod()*. Si une autre exception est levée par le processus du gestionnaire, elle est convertie en une `RemoteError` et est levée par *_callmethod()*.

Notez en particulier qu'une exception est levée si *methodname* n'est pas *exposée*.

Un exemple d'utilisation de *_callmethod()* :

```
>>> l = manager.list(range(10))
>>> l._callmethod('__len__')
10
>>> l._callmethod('__getitem__', (slice(2, 7),)) # equivalent to l[2:7]
[2, 3, 4, 5, 6]
>>> l._callmethod('__getitem__', (20,))          # equivalent to l[20]
Traceback (most recent call last):
...
IndexError: list index out of range
```

_getvalue()

Renvoie une copie du référent.

Si le référent n'est pas sérialisable, une exception est levée.

__repr__()

Renvoie la représentation de l'objet mandataire.

__str__()

Renvoie la représentation du référent.

Nettoyage

Un mandataire utilise un *callback* sous une référence faible de façon à ce que quand il est collecté par le ramasse-miettes, il se désenregistre auprès du gestionnaire qui possède le référent.

Un objet partagé est supprimé par le processus gestionnaire quand plus aucun mandataire ne le référence.

Pools de processus

On peut créer un pool de processus qui exécuteront les tâches qui lui seront soumises avec la classe `Pool`.

class `multiprocessing.pool.`**`Pool`**(*[processes[, initializer[, initargs[, maxtasksperchild[, context]]]]]*)

Un objet *process pool* qui contrôle un pool de processus *workers* auquel sont soumises des tâches. Il supporte les résultats asynchrones avec des *timeouts* et des *callbacks* et possède une implémentation parallèle de *map*.

processes is the number of worker processes to use. If *processes* is `None` then the number returned by `os.process_cpu_count()` is used.

Si *initializer* n'est pas `None`, chaque processus *worker* appellera `initializer(*initargs)` en démarrant.

maxtasksperchild est le nombre de tâches qu'un processus *worker* peut accomplir avant de se fermer et d'être remplacé par un *worker* frais, pour permettre aux ressources inutilisées d'être libérées. Par défaut *maxtasksperchild* est `None`, ce qui signifie que le *worker* vit aussi longtemps que le pool.

context peut être utilisé pour préciser le contexte utilisé pour démarrer les processus *workers*. Habituellement un pool est créé à l'aide de la fonction `multiprocessing.Pool()` ou de la méthode `Pool()` d'un objet de contexte. Dans les deux cas *context* est réglé de façon appropriée.

Notez que les méthodes de l'objet *pool* ne doivent être appelées que par le processus qui l'a créé.

> **Avertissement**
>
> Les objets `multiprocessing.pool` ont des ressources internes qui doivent être correctement gérées (comme toute autre ressource) en utilisant le pool comme gestionnaire de contexte ou en appelant `close()` et `terminate()` manuellement. Si cela n'est pas fait, le processus peut être bloqué à la finalisation.
>
> Notez qu'il n'est **pas** correct de compter sur le ramasse-miette pour détruire le pool car CPython ne garantit pas que le *finalizer* du pool est appelé (voir `object.__del__()` pour plus d'informations).

Modifié dans la version 3.2 : Added the *maxtasksperchild* parameter.

Modifié dans la version 3.4 : Added the *context* parameter.

Modifié dans la version 3.13 : *processes* uses `os.process_cpu_count()` by default, instead of `os.cpu_count()`.

> **Note**
>
> Les processus *workers* à l'intérieur d'une `Pool` vivent par défaut aussi longtemps que la file de travail du pool. Un modèle fréquent chez d'autres systèmes (tels qu'Apache, *mod_wsgi*, etc.) pour libérer les ressources détenues par les *workers* est d'autoriser un *worker* dans le pool à accomplir seulement une certaine charge de travail avant de se fermer, se retrouvant nettoyé et remplacé par un nouveau processus fraîchement lancé. L'argument *maxtasksperchild* de `Pool` expose cette fonctionnalité à l'utilisateur final.

apply(*func[, args[, kwds]]*)

Appelle *func* avec les arguments *args* et les arguments nommés *kwds*. Bloque jusqu'à ce que que le résultat soit prêt. En raison de ce blocage, `apply_async()` est préférable pour exécuter du travail en parallèle. De plus, *func* est exécutée sur un seul des *workers* du pool.

apply_async(*func[, args[, kwds[, callback[, error_callback]]]]*)

Une variante de la méthode `apply()` qui renvoie un objet `AsyncResult`.

Si *callback* est précisé alors ce doit être un objet appelable qui accepte un seul argument. Quand le résultat est prêt, *callback* est appelé avec ce résultat, si l'appel n'échoue pas auquel cas *error_callback* est appelé à la place.

Si *error_callback* est précisé alors ce doit être un objet appelable qui accepte un seul argument. Si la fonction cible échoue, alors *error_callback* est appelé avec l'instance de l'exception.

Les *callbacks* doivent se terminer immédiatement, autrement le fil d'exécution qui gère les résultats se retrouverait bloqué.

map(*func*, *iterable*[, *chunksize*])

Un équivalent parallèle de la fonction native `map()` (qui ne prend qu'un seul argument *itérable* ; pour en passer plusieurs, référez-vous à `starmap()`). Elle bloque jusqu'à ce que le résultat soit prêt.

La méthode découpe l'itérable en un nombre de morceaux qu'elle envoie au pool de processus comme des tâches séparées. La taille (approximative) de ces morceaux peut être précisée en passant à *chunksize* un entier positif.

Notez que cela peut entraîner une grosse consommation de mémoire pour les itérables très longs. Envisagez d'utiliser `imap()` ou `imap_unordered()` avec l'option *chunksize* explicite pour une meilleure efficacité.

map_async(*func*, *iterable*[, *chunksize*[, *callback*[, *error_callback*]]])

Une variante de la méthode `map()` qui renvoie un objet `AsyncResult`.

Si *callback* est précisé alors ce doit être un objet appelable qui accepte un seul argument. Quand le résultat est prêt, *callback* est appelé avec ce résultat, si l'appel n'échoue pas auquel cas *error_callback* est appelé à la place.

Si *error_callback* est précisé alors ce doit être un objet appelable qui accepte un seul argument. Si la fonction cible échoue, alors *error_callback* est appelé avec l'instance de l'exception.

Les *callbacks* doivent se terminer immédiatement, autrement le fil d'exécution qui gère les résultats se retrouverait bloqué.

imap(*func*, *iterable*[, *chunksize*])

Une version paresseuse de `map()`.

L'argument *chunksize* est le même que celui utilisé par la méthode `map()`. Pour de très longs itérables, utiliser une grande valeur pour *chunksize* peut faire s'exécuter la tâche **beaucoup** plus rapidement qu'en utilisant la valeur par défaut de `1`.

Aussi, si *chucksize* vaut `1` alors la méthode `next()` de l'itérateur renvoyé par `imap()` prend un paramètre optionnel *timeout* : `next(timeout)` lève une `multiprocessing.TimeoutError` si le résultat ne peut pas être renvoyé avant *timeout* secondes.

imap_unordered(*func*, *iterable*[, *chunksize*])

Identique à `imap()` si ce n'est que l'ordre des résultats de l'itérateur renvoyé doit être considéré comme arbitraire. (L'ordre n'est garanti que quand il n'y a qu'un *worker*.)

starmap(*func*, *iterable*[, *chunksize*])

Like `map()` except that the elements of the *iterable* are expected to be iterables that are unpacked as arguments.

Par conséquent un *iterable* `[(1,2), (3, 4)]` donnera pour résultat `[func(1,2), func(3,4)]`.

Ajouté dans la version 3.3.

starmap_async(*func*, *iterable*[, *chunksize*[, *callback*[, *error_callback*]]])

Une combinaison de `starmap()` et `map_async()` qui itère sur *iterable* (composé d'itérables) et appelle *func* pour chaque itérable dépaqueté. Renvoie l'objet résultat.

Ajouté dans la version 3.3.

close()

Empêche de nouvelles tâches d'être envoyées à la *pool*. Les processus *workers* se terminent une fois que

toutes les tâches ont été complétées.

terminate()

Stoppe immédiatement les processus *workers* sans finaliser les travaux courants. Quand l'objet *pool* est collecté par le ramasse-miettes, sa méthode `terminate()` est appelée immédiatement.

join()

Attend que les processus *workers* se terminent. Il est nécessaire d'appeler `close()` ou `terminate()` avant d'utiliser `join()`.

Modifié dans la version 3.3 : Les pools de *workers* supportent maintenant le protocole des gestionnaires de contexte – voir *Le type gestionnaire de contexte*. `__enter__()` renvoie l'objet *pool* et `__exit__()` appelle `terminate()`.

class multiprocessing.pool.**AsyncResult**

La classe des résultats renvoyés par `Pool.apply_async()` et `Pool.map_async()`.

get([*timeout*])

Renvoie le résultat quand il arrive. Si *timeout* n'est pas `None` et que le résultat n'arrive pas avant *timeout* secondes, une `multiprocessing.TimeoutError` est levée. Si l'appel distance lève une exception, alors elle est relayée par `get()`.

wait([*timeout*])

Attend que le résultat soit disponible ou que *timeout* secondes s'écoulent.

ready()

Renvoie `True` ou `False` suivant si la tâche est accomplie.

successful()

Renvoie `True` ou `False` suivant si la tâche est accomplie sans lever d'exception. Lève une `ValueError` si le résultat n'est pas prêt.

Modifié dans la version 3.7 : Si le résultat n'est pas prêt, une `ValueError` est levée au lieu d'une `AssertionError` auparavant.

Les exemples suivants présentent l'utilisation d'un pool de *workers* :

```
from multiprocessing import Pool
import time

def f(x):
    return x*x

if __name__ == '__main__':
    with Pool(processes=4) as pool:          # start 4 worker processes
        result = pool.apply_async(f, (10,)) # evaluate "f(10)" asynchronously in a␣
↪single process
        print(result.get(timeout=1))         # prints "100" unless your computer is␣
↪*very* slow

        print(pool.map(f, range(10)))        # prints "[0, 1, 4,..., 81]"

        it = pool.imap(f, range(10))
        print(next(it))                      # prints "0"
        print(next(it))                      # prints "1"
        print(it.next(timeout=1))            # prints "4" unless your computer is␣
↪*very* slow

        result = pool.apply_async(time.sleep, (10,))
        print(result.get(timeout=1))         # raises multiprocessing.TimeoutError
```

Auditeurs et Clients

Habituellement l'échange de messages entre processus est réalisé en utilisant des files ou des objets `Connection` renvoyés par `Pipe()`.

Cependant, le module `multiprocessing.connection` permet un peu plus de flexibilité. Il fournit un message de plus haut-niveau orienté API pour gérer des connecteurs ou des tubes nommés sous Windows. Il gère aussi l'authentification par condensat (*digest authentication* en anglais) en utilisant le module `hmac`, et pour interroger de multiples connexions en même temps.

`multiprocessing.connection.`**`deliver_challenge`**(*connection*, *authkey*)

Envoie un message généré aléatoirement à l'autre extrémité de la connexion et attend une réponse.

Si la réponse correspond au condensat du message avec la clé *authkey*, alors un message de bienvenue est envoyé à l'autre extrémité de la connexion. Autrement, une `AuthenticationError` est levée.

`multiprocessing.connection.`**`answer_challenge`**(*connection*, *authkey*)

Reçoit un message, calcule le condensat du message en utilisant la clé *authkey*, et envoie le condensat en réponse.

Si un message de bienvenue n'est pas reçu, une `AuthenticationError` est levée.

`multiprocessing.connection.`**`Client`**(*address*[, *family*[, *authkey*]])

Essaie d'établir une connexion avec l'auditeur qui utilise l'adresse *address*, renvoie une `Connection`.

Le type de la connexion est déterminé par l'argument *family*, mais il peut généralement être omis puisqu'il peut être inféré depuis le format d'*address*. (Voir *Formats d'adresses*)

If *authkey* is given and not `None`, it should be a byte string and will be used as the secret key for an HMAC-based authentication challenge. No authentication is done if *authkey* is `None`. `AuthenticationError` is raised if authentication fails. See *Clés d'authentification*.

`class` `multiprocessing.connection.`**`Listener`**([*address*[, *family*[, *backlog*[, *authkey*]]]])

Une enveloppe autour d'un connecteur lié ou un tube nommé sous Windows qui écoute pour des connexions.

address est l'adresse à utiliser par le connecteur lié ou le tube nommé de l'objet auditeur.

> **Note**
>
> Si une adresse '0.0.0.0' est utilisée, l'adresse ne sera pas un point d'accès connectable sous Windows. Si vous avez besoin d'un point d'accès connectable, utilisez '127.0.0.1'.

family est le type de connecteur (ou tube nommé) à utiliser. Cela peut être l'une des chaînes `'AF_INET'` (pour un connecteur TCP), `'AF_UNIX'` (pour un connecteur Unix) ou `'AF_PIPE'` (pour un tube nommé sous Windows). Seulement le premier d'entre eux est garanti d'être disponible. Si *family* est `None`, la famille est inférée depuis le format d'*address*. Si *address* est aussi `None`, la famille par défaut est utilisée. La famille par défaut est supposée être la plus rapide disponible. Voir *Formats d'adresses*. Notez que si la *family* est `'AF_UNIX'` et qu'*address* est `None`, le connecteur est créé dans un répertoire temporaire privé créé avec `tempfile.mkstemp()`.

Si l'objet auditeur utilise un connecteur alors *backlog* (1 par défaut) est passé à la méthode `listen()` du connecteur une fois qu'il a été lié.

If *authkey* is given and not `None`, it should be a byte string and will be used as the secret key for an HMAC-based authentication challenge. No authentication is done if *authkey* is `None`. `AuthenticationError` is raised if authentication fails. See *Clés d'authentification*.

`accept`()

Accepte une connexion sur le connecteur lié ou le tube nommé de l'objet auditeur et renvoie un objet `Connection`. Si la tentative d'authentification échoue, une `AuthenticationError` est levée.

`close`()

Ferme le connecteur lié ou le tube nommé de l'objet auditeur. La méthode est appelée automatiquement

quand l'auditeur est collecté par le ramasse-miettes. Il est cependant conseillé de l'appeler explicitement.

Les objets auditeurs ont aussi les propriétés en lecture seule suivantes :

address

L'adresse utilisée par l'objet auditeur.

last_accepted

L'adresse depuis laquelle a été établie la dernière connexion. `None` si aucune n'est disponible.

Modifié dans la version 3.3 : Les objets auditeurs supportent maintenant le protocole des gestionnaires de contexte – voir *Le type gestionnaire de contexte*. `__enter__()` renvoie l'objet auditeur, et `__exit__()` appelle `close()`.

`multiprocessing.connection.`**`wait`**(*object_list*, *timeout=None*)

Attend qu'un objet d'*object_list* soit prêt. Renvoie la liste de ces objets d'*object_list* qui sont prêts. Si *timeout* est un nombre flottant, l'appel bloquera au maximum ce nombre de secondes. Si *timeout* est `None`, l'appelle bloquera pour une durée non limitée. Un *timeout* négatif est équivalent à un *timeout* nul.

For both POSIX and Windows, an object can appear in *object_list* if it is
— un objet `Connection` accessible en lecture ;
— un objet `socket.socket` connecté et accessible en lecture ; ou
— l'attribut `sentinel` d'un objet `Process`.
Une connexion (*socket* en anglais) est prête quand il y a des données disponibles en lecture dessus, ou que l'autre extrémité a été fermée.

POSIX : `wait(object_list, timeout)` almost equivalent `select.select(object_list, [], [], timeout)`. The difference is that, if `select.select()` is interrupted by a signal, it can raise `OSError` with an error number of `EINTR`, whereas `wait()` will not.

Windows : An item in *object_list* must either be an integer handle which is waitable (according to the definition used by the documentation of the Win32 function `WaitForMultipleObjects()`) or it can be an object with a `fileno()` method which returns a socket handle or pipe handle. (Note that pipe handles and socket handles are **not** waitable handles.)

Ajouté dans la version 3.3.

Exemples

Le code serveur suivant crée un auditeur qui utilise `'secret password'` comme clé d'authentification. Il attend ensuite une connexion et envoie les données au client :

```
from multiprocessing.connection import Listener
from array import array

address = ('localhost', 6000)     # family is deduced to be 'AF_INET'

with Listener(address, authkey=b'secret password') as listener:
    with listener.accept() as conn:
        print('connection accepted from', listener.last_accepted)

        conn.send([2.25, None, 'junk', float])

        conn.send_bytes(b'hello')

        conn.send_bytes(array('i', [42, 1729]))
```

Le code suivant se connecte au serveur et en reçoit des données :

```
from multiprocessing.connection import Client
from array import array

address = ('localhost', 6000)
```

(suite sur la page suivante)

(suite de la page précédente)

```
with Client(address, authkey=b'secret password') as conn:
    print(conn.recv())                  # => [2.25, None, 'junk', float]

    print(conn.recv_bytes())            # => 'hello'

    arr = array('i', [0, 0, 0, 0, 0])
    print(conn.recv_bytes_into(arr))    # => 8
    print(arr)                          # => array('i', [42, 1729, 0, 0, 0])
```

Le code suivant utilise `wait()` pour attendre des messages depuis plusieurs processus à la fois :

```
from multiprocessing import Process, Pipe, current_process
from multiprocessing.connection import wait

def foo(w):
    for i in range(10):
        w.send((i, current_process().name))
    w.close()

if __name__ == '__main__':
    readers = []

    for i in range(4):
        r, w = Pipe(duplex=False)
        readers.append(r)
        p = Process(target=foo, args=(w,))
        p.start()
        # We close the writable end of the pipe now to be sure that
        # p is the only process which owns a handle for it.  This
        # ensures that when p closes its handle for the writable end,
        # wait() will promptly report the readable end as being ready.
        w.close()

    while readers:
        for r in wait(readers):
            try:
                msg = r.recv()
            except EOFError:
                readers.remove(r)
            else:
                print(msg)
```

Formats d'adresses

- une adresse `'AF_INET'` est une paire de la forme `(hostname, port)` où *hostname* est une chaîne et *port* un entier ;
- une adresse `'AF_UNIX'` est une chaîne représentant un nom de fichier sur le système de fichiers ;
- An `'AF_PIPE'` address is a string of the form `r'\\.\pipe\PipeName'`. To use `Client()` to connect to a named pipe on a remote computer called *ServerName* one should use an address of the form `r'\\ServerName\pipe\PipeName'` instead.

Notez que toute chaîne commençant par deux antislashs est considérée par défaut comme l'adresse d'un `'AF_PIPE'` plutôt qu'une adresse `'AF_UNIX'`.

Clés d'authentification

Quand `Connection.recv` est utilisée, les données reçues sont automatiquement désérialisées par *pickle*. Malheureusement désérialiser des données depuis une source non sûre constitue un risque de sécurité. Par conséquent `Listener` et `Client()` utilisent le module `hmac` pour fournir une authentification par condensat.

Une clé d'authentification est une chaîne d'octets qui peut être vue comme un mot de passe : quand une connexion est établie, les deux interlocuteurs vont demander à l'autre une preuve qu'il connaît la clé d'authentification. (Démontrer que les deux utilisent la même clé n'implique **pas** d'échanger la clé sur la connexion.)

Si l'authentification est requise et qu'aucune clé n'est spécifiée alors la valeur de retour de `current_process().authkey` est utilisée (voir `Process`). Celle valeur est automatiquement héritée par tout objet `Process` créé par le processus courant. Cela signifie que (par défaut) tous les processus d'un programme multi-processus partageront une clé d'authentification unique qui peut être utilisée pour mettre en place des connexions entre-eux.

Des clés d'authentification adaptées peuvent aussi être générées par `os.urandom()`.

Journalisation

Un certain support de la journalisation est disponible. Notez cependant que le le paquet `logging` n'utilise pas de verrous partagés entre les processus et il est donc possible (dépendant du type de gestionnaire) que les messages de différents processus soient mélangés.

`multiprocessing.`**`get_logger`**`()`

Renvoie le journaliseur utilisé par `multiprocessing`. Si nécessaire, un nouveau sera créé.

When first created the logger has level `logging.NOTSET` and no default handler. Messages sent to this logger will not by default propagate to the root logger.

Notez que sous Windows les processus fils n'hériteront que du niveau du journaliseur du processus parent – toute autre personnalisation du journaliseur ne sera pas héritée.

`multiprocessing.`**`log_to_stderr`**`(`*level=None*`)`

This function performs a call to `get_logger()` but in addition to returning the logger created by get_logger, it adds a handler which sends output to `sys.stderr` using format `'[%(levelname)s/%(processName)s] %(message)s'`. You can modify `levelname` of the logger by passing a `level` argument.

L'exemple ci-dessous présente une session avec la journalisation activée :

```
>>> import multiprocessing, logging
>>> logger = multiprocessing.log_to_stderr()
>>> logger.setLevel(logging.INFO)
>>> logger.warning('doomed')
[WARNING/MainProcess] doomed
>>> m = multiprocessing.Manager()
[INFO/SyncManager-...] child process calling self.run()
[INFO/SyncManager-...] created temp directory /.../pymp-...
[INFO/SyncManager-...] manager serving at '/.../listener-...'
>>> del m
[INFO/MainProcess] sending shutdown message to manager
[INFO/SyncManager-...] manager exiting with exitcode 0
```

Pour un tableau complet des niveaux de journalisation, voir le module `logging`.

Le module `multiprocessing.dummy`

`multiprocessing.dummy` réplique toute l'API de `multiprocessing` mais n'est rien de plus qu'une interface autour du module `threading`.

En particulier, la fonction `Pool` du module `multiprocessing.dummy` renvoie une instance de `ThreadPool`, qui est une sous-classe de `Pool`. Elle a la même interface, mais elle utilise un pool de fils d'exécution plutôt qu'un pool de processus.

class multiprocessing.pool.**ThreadPool**([*processes*[, *initializer*[, *initargs*]]])

Un objet qui contrôle un pool de fils d'exécution auquel des tâches peuvent être envoyées. L'interface des instances de `ThreadPool` est entièrement compatible avec celle des instances de `Pool`, et leur ressources doivent être aussi correctement gérées, soit en utilisant le pool avec un contexte, soit en appelant explicitement `close()` et `terminate()`.

processes is the number of worker threads to use. If *processes* is `None` then the number returned by `os.process_cpu_count()` is used.

Si *initializer* n'est pas `None`, chaque processus *worker* appellera `initializer(*initargs)` en démarrant.

À la différence de `Pool`, *maxtasksperchild* et *context* ne peuvent pas être passés en arguments.

> **Note**
>
> La classe `ThreadPool` a la même interface que la classe `Pool`, dont l'implémentation repose sur un pool de processus, et a été introduite avant le module `concurrent.futures`. Par conséquent elle implémente des opérations qui n'ont pas vraiment de sens pour un pool implémenté avec des fils d'exécution et possède son propre type pour représenter le statut de tâches asynchrones, `AsyncResult`, qui n'est pas géré par les autres modules.
>
> Il est souvent plus judicieux d'utiliser `concurrent.futures.ThreadPoolExecutor` qui a une interface plus simple, qui a été pensée dès l'origine pour les fils d'exécution et qui renvoie des instances de `concurrent.futures.Future` qui sont compatibles avec de nombreux modules, dont `asyncio`.

18.2.3 Lignes directrices de programmation

Il y a certaines lignes directrices et idiomes à respecter pour utiliser `multiprocessing`.

Toutes les méthodes de démarrage

Les règles suivantes s'appliquent aux méthodes de démarrage.

Éviter les états partagés

Autant que possible, il faut éviter de transférer de gros volumes de données entre les processus.

Il est souvent plus judicieux de se borner à utiliser des files et des tubes pour gérer la communication entre processus plutôt que d'utiliser des primitives de synchronisation plus bas-niveau.

Sérialisation

Assurez-vous que les arguments passés aux méthodes des mandataires soient sérialisables (*pickables*).

Sûreté des mandataires à travers les fils d'exécution

N'utilisez pas d'objet mandataire depuis plus d'un fil d'exécution à moins que vous ne le protégiez avec un verrou.

Il n'y a jamais de problème à avoir plusieurs processus qui utilisent un *même* mandataire.

Attendre les processus zombies

On POSIX when a process finishes but has not been joined it becomes a zombie. There should never be very many because each time a new process starts (or `active_children()` is called) all completed processes which have not yet been joined will be joined. Also calling a finished process's `Process.is_alive` will join the process. Even so it is probably good practice to explicitly join all the processes that you start.

Mieux vaut hériter que sérialiser - désérialiser

Quand vous utilisez les méthodes de démarrage *spawn* ou *forkserver*, de nombreux types de `multiprocessing` nécessitent d'être sérialisés pour que les processus fils puissent les utiliser. Cependant, il faut généralement éviter d'envoyer des objets partagés aux autres processus en utilisant des tubes ou des files. Vous devez plutôt vous arranger pour qu'un processus qui nécessite l'accès à une ressource partagée créée autre part qu'il en hérite depuis un de ses processus ancêtres.

Éviter de terminer les processus

Utiliser la méthode `Process.terminate` pour stopper un processus risque de casser ou de rendre indisponible aux autres processus des ressources partagées (comme des verrous, sémaphores, tubes et files) actuellement utilisées par le processus.

Il est donc préférable de n'utiliser `Process.terminate` que sur les processus qui n'utilisent jamais de ressources partagées.

Attendre les processus qui utilisent des files

Gardez à l'esprit qu'un processus qui a placé des éléments dans une file attend que tous les éléments mis en tampon soient consommés par le fil d'exécution « chargeur » du tube sous-jacent avant de se terminer (le processus fils peut appeler la méthode `Queue.cancel_join_thread` de la queue pour éviter ce comportement).

Cela signifie que chaque fois que vous utilisez une file, vous devez vous assurer que tous les éléments qui y ont été placés ont été effectivement supprimés avant que le processus ne soit attendu. Autrement vous ne pouvez pas être sûr que les processus qui ont placé des éléments dans la file se termineront. Souvenez-vous aussi que tous les processus non *daemons* sont attendus automatiquement.

L'exemple suivant provoque un interblocage :

```
from multiprocessing import Process, Queue

def f(q):
    q.put('X' * 1000000)

if __name__ == '__main__':
    queue = Queue()
    p = Process(target=f, args=(queue,))
    p.start()
    p.join()                    # this deadlocks
    obj = queue.get()
```

Une solution ici consiste à intervertir les deux dernières lignes (ou simplement à supprimer la ligne `p.join()`).

Passer explicitement les ressources aux processus fils

On POSIX using the *fork* start method, a child process can make use of a shared resource created in a parent process using a global resource. However, it is better to pass the object as an argument to the constructor for the child process.

En plus de rendre le code (potentiellement) compatible avec Windows et les autres méthodes de démarrage, cela assure aussi que tant que le processus fils est en vie, l'objet ne sera pas collecté par le ramasse-miettes du processus parent. Cela peut être important si certaines ressources sont libérées quand l'objet est collecté par le ramasse-miettes du processus parent.

Donc par exemple

```
from multiprocessing import Process, Lock

def f():
    ... do something using "lock" ...

if __name__ == '__main__':
    lock = Lock()
    for i in range(10):
        Process(target=f).start()
```

devrait être réécrit comme

```
from multiprocessing import Process, Lock
```

(suite sur la page suivante)

(suite de la page précédente)

```
def f(l):
    ... do something using "l" ...

if __name__ == '__main__':
    lock = Lock()
    for i in range(10):
        Process(target=f, args=(lock,)).start()
```

Faire attention à remplacer `sys.stdin` par un objet simili-fichier

À l'origine, `multiprocessing` appelait inconditionnellement :

```
os.close(sys.stdin.fileno())
```

dans la méthode `multiprocessing.Process._bootstrap()` — cela provoquait des problèmes avec les processus imbriqués. Cela peut être changé en :

```
sys.stdin.close()
sys.stdin = open(os.open(os.devnull, os.O_RDONLY), closefd=False)
```

Which solves the fundamental issue of processes colliding with each other resulting in a bad file descriptor error, but introduces a potential danger to applications which replace `sys.stdin()` with a "file-like object" with output buffering. This danger is that if multiple processes call `close()` on this file-like object, it could result in the same data being flushed to the object multiple times, resulting in corruption.

Si vous écrivez un objet simili-fichier et implémentez votre propre cache, vous pouvez le rendre sûr pour les *forks* en stockant le *pid* chaque fois que vous ajoutez des données au cache, et annulez le cache quand le *pid* change. Par exemple :

```
@property
def cache(self):
    pid = os.getpid()
    if pid != self._pid:
        self._pid = pid
        self._cache = []
    return self._cache
```

Pour plus d'informations, voir bpo-5155, bpo-5313 et bpo-5331

Les méthodes de démarrage *spawn* et *forkserver*

There are a few extra restrictions which don't apply to the *fork* start method.

Contraintes supplémentaires sur la sérialisation

Assurez-vous que tous les argument de `Process.__init__()` sont sérialisables avec *pickle*. Aussi, si vous héritez de `Process`, assurez-vous que toutes les instances sont sérialisables quand la méthode `Process.start` est appelée.

Variables globales

Gardez en tête que si le code exécuté dans un processus fils essaie d'accéder à une variable globale, alors la valeur qu'il voit (s'il y en a une) pourrait ne pas être la même que la valeur du processus parent au moment même où `Process.start` est appelée.

Cependant, les variables globales qui sont juste des constantes de modules ne posent pas de problèmes.

Importation sécurisée du module principal

Make sure that the main module can be safely imported by a new Python interpreter without causing unintended side effects (such as starting a new process).

Par exemple, utiliser la méthode de démarrage *spawn* ou *forkserver* pour lancer le module suivant échouerait avec une `RuntimeError` :

```
from multiprocessing import Process

def foo():
    print('hello')

p = Process(target=foo)
p.start()
```

Vous devriez plutôt protéger le « point d'entrée » du programme en utilisant `if __name__ == '__main__'` : comme suit :

```
from multiprocessing import Process, freeze_support, set_start_method

def foo():
    print('hello')

if __name__ == '__main__':
    freeze_support()
    set_start_method('spawn')
    p = Process(target=foo)
    p.start()
```

(La ligne `freeze_support()` peut être omise si le programme est uniquement lancé normalement et pas figé.)

Cela permet aux interpréteurs Python fraîchement instanciés d'importer en toute sécurité le module et d'exécution ensuite la fonction `foo()`.

Des restrictions similaires s'appliquent si un pool ou un gestionnaire est créé dans le module principal.

18.2.4 Exemples

Démonstration de comment créer et utiliser des gestionnaires et mandataires personnalisés :

```
from multiprocessing import freeze_support
from multiprocessing.managers import BaseManager, BaseProxy
import operator

##

class Foo:
    def f(self):
        print('you called Foo.f()')
    def g(self):
        print('you called Foo.g()')
    def _h(self):
        print('you called Foo._h()')

# A simple generator function
def baz():
    for i in range(10):
        yield i*i

# Proxy type for generator objects
class GeneratorProxy(BaseProxy):
    _exposed_ = ['__next__']
    def __iter__(self):
        return self
```

(suite sur la page suivante)

(suite de la page précédente)

```
    def __next__(self):
        return self._callmethod('__next__')

# Function to return the operator module
def get_operator_module():
    return operator

##

class MyManager(BaseManager):
    pass

# register the Foo class; make `f()` and `g()` accessible via proxy
MyManager.register('Foo1', Foo)

# register the Foo class; make `g()` and `_h()` accessible via proxy
MyManager.register('Foo2', Foo, exposed=('g', '_h'))

# register the generator function baz; use `GeneratorProxy` to make proxies
MyManager.register('baz', baz, proxytype=GeneratorProxy)

# register get_operator_module(); make public functions accessible via proxy
MyManager.register('operator', get_operator_module)

##

def test():
    manager = MyManager()
    manager.start()

    print('-' * 20)

    f1 = manager.Foo1()
    f1.f()
    f1.g()
    assert not hasattr(f1, '_h')
    assert sorted(f1._exposed_) == sorted(['f', 'g'])

    print('-' * 20)

    f2 = manager.Foo2()
    f2.g()
    f2._h()
    assert not hasattr(f2, 'f')
    assert sorted(f2._exposed_) == sorted(['g', '_h'])

    print('-' * 20)

    it = manager.baz()
    for i in it:
        print('<%d>' % i, end=' ')
    print()

    print('-' * 20)

    op = manager.operator()
```

(suite sur la page suivante)

(suite de la page précédente)

```
    print('op.add(23, 45) =', op.add(23, 45))
    print('op.pow(2, 94) =', op.pow(2, 94))
    print('op._exposed_ =', op._exposed_)

##

if __name__ == '__main__':
    freeze_support()
    test()
```

En utilisant `Pool` :

```
import multiprocessing
import time
import random
import sys

#
# Functions used by test code
#

def calculate(func, args):
    result = func(*args)
    return '%s says that %s%s = %s' % (
        multiprocessing.current_process().name,
        func.__name__, args, result
        )

def calculatestar(args):
    return calculate(*args)

def mul(a, b):
    time.sleep(0.5 * random.random())
    return a * b

def plus(a, b):
    time.sleep(0.5 * random.random())
    return a + b

def f(x):
    return 1.0 / (x - 5.0)

def pow3(x):
    return x ** 3

def noop(x):
    pass

#
# Test code
#

def test():
    PROCESSES = 4
    print('Creating pool with %d processes\n' % PROCESSES)
```

(suite sur la page suivante)

(suite de la page précédente)

```
    with multiprocessing.Pool(PROCESSES) as pool:
        #
        # Tests
        #

        TASKS = [(mul, (i, 7)) for i in range(10)] + \
                [(plus, (i, 8)) for i in range(10)]

        results = [pool.apply_async(calculate, t) for t in TASKS]
        imap_it = pool.imap(calculatestar, TASKS)
        imap_unordered_it = pool.imap_unordered(calculatestar, TASKS)

        print('Ordered results using pool.apply_async():')
        for r in results:
            print('\t', r.get())
        print()

        print('Ordered results using pool.imap():')
        for x in imap_it:
            print('\t', x)
        print()

        print('Unordered results using pool.imap_unordered():')
        for x in imap_unordered_it:
            print('\t', x)
        print()

        print('Ordered results using pool.map() --- will block till complete:')
        for x in pool.map(calculatestar, TASKS):
            print('\t', x)
        print()

        #
        # Test error handling
        #

        print('Testing error handling:')

        try:
            print(pool.apply(f, (5,)))
        except ZeroDivisionError:
            print('\tGot ZeroDivisionError as expected from pool.apply()')
        else:
            raise AssertionError('expected ZeroDivisionError')

        try:
            print(pool.map(f, list(range(10))))
        except ZeroDivisionError:
            print('\tGot ZeroDivisionError as expected from pool.map()')
        else:
            raise AssertionError('expected ZeroDivisionError')

        try:
            print(list(pool.imap(f, list(range(10)))))
        except ZeroDivisionError:
            print('\tGot ZeroDivisionError as expected from list(pool.imap())')
```

(suite sur la page suivante)

(suite de la page précédente)

```
        else:
            raise AssertionError('expected ZeroDivisionError')

        it = pool.imap(f, list(range(10)))
        for i in range(10):
            try:
                x = next(it)
            except ZeroDivisionError:
                if i == 5:
                    pass
            except StopIteration:
                break
            else:
                if i == 5:
                    raise AssertionError('expected ZeroDivisionError')

        assert i == 9
        print('\tGot ZeroDivisionError as expected from IMapIterator.next()')
        print()

        #
        # Testing timeouts
        #

        print('Testing ApplyResult.get() with timeout:', end=' ')
        res = pool.apply_async(calculate, TASKS[0])
        while 1:
            sys.stdout.flush()
            try:
                sys.stdout.write('\n\t%s' % res.get(0.02))
                break
            except multiprocessing.TimeoutError:
                sys.stdout.write('.')
        print()
        print()

        print('Testing IMapIterator.next() with timeout:', end=' ')
        it = pool.imap(calculatestar, TASKS)
        while 1:
            sys.stdout.flush()
            try:
                sys.stdout.write('\n\t%s' % it.next(0.02))
            except StopIteration:
                break
            except multiprocessing.TimeoutError:
                sys.stdout.write('.')
        print()
        print()


if __name__ == '__main__':
    multiprocessing.freeze_support()
    test()
```

Un exemple montrant comment utiliser des files pour alimenter en tâches une collection de processus *workers* et collecter les résultats :

```
import time
import random

from multiprocessing import Process, Queue, current_process, freeze_support

#
# Function run by worker processes
#

def worker(input, output):
    for func, args in iter(input.get, 'STOP'):
        result = calculate(func, args)
        output.put(result)

#
# Function used to calculate result
#

def calculate(func, args):
    result = func(*args)
    return '%s says that %s%s = %s' % \
        (current_process().name, func.__name__, args, result)

#
# Functions referenced by tasks
#

def mul(a, b):
    time.sleep(0.5*random.random())
    return a * b

def plus(a, b):
    time.sleep(0.5*random.random())
    return a + b

#
#
#

def test():
    NUMBER_OF_PROCESSES = 4
    TASKS1 = [(mul, (i, 7)) for i in range(20)]
    TASKS2 = [(plus, (i, 8)) for i in range(10)]

    # Create queues
    task_queue = Queue()
    done_queue = Queue()

    # Submit tasks
    for task in TASKS1:
        task_queue.put(task)

    # Start worker processes
    for i in range(NUMBER_OF_PROCESSES):
        Process(target=worker, args=(task_queue, done_queue)).start()

    # Get and print results
```

(suite sur la page suivante)

(suite de la page précédente)

```
    print('Unordered results:')
    for i in range(len(TASKS1)):
        print('\t', done_queue.get())

    # Add more tasks using `put()`
    for task in TASKS2:
        task_queue.put(task)

    # Get and print some more results
    for i in range(len(TASKS2)):
        print('\t', done_queue.get())

    # Tell child processes to stop
    for i in range(NUMBER_OF_PROCESSES):
        task_queue.put('STOP')


if __name__ == '__main__':
    freeze_support()
    test()
```

18.3 multiprocessing.shared_memory --- Shared memory for direct access across processes

Code source : Lib/multiprocessing/shared_memory.py

Ajouté dans la version 3.8.

This module provides a class, *SharedMemory*, for the allocation and management of shared memory to be accessed by one or more processes on a multicore or symmetric multiprocessor (SMP) machine. To assist with the life-cycle management of shared memory especially across distinct processes, a *BaseManager* subclass, *SharedMemoryManager*, is also provided in the *multiprocessing.managers* module.

In this module, shared memory refers to "POSIX style" shared memory blocks (though is not necessarily implemented explicitly as such) and does not refer to "distributed shared memory". This style of shared memory permits distinct processes to potentially read and write to a common (or shared) region of volatile memory. Processes are conventionally limited to only have access to their own process memory space but shared memory permits the sharing of data between processes, avoiding the need to instead send messages between processes containing that data. Sharing data directly via memory can provide significant performance benefits compared to sharing data via disk or socket or other communications requiring the serialization/deserialization and copying of data.

class multiprocessing.shared_memory.**SharedMemory**(*name=None*, *create=False*, *size=0*, ***, *track=True*)

Create an instance of the `SharedMemory` class for either creating a new shared memory block or attaching to an existing shared memory block. Each shared memory block is assigned a unique name. In this way, one process can create a shared memory block with a particular name and a different process can attach to that same shared memory block using that same name.

As a resource for sharing data across processes, shared memory blocks may outlive the original process that created them. When one process no longer needs access to a shared memory block that might still be needed by other processes, the *close()* method should be called. When a shared memory block is no longer needed by any process, the *unlink()* method should be called to ensure proper cleanup.

Paramètres

— **name** (*str | None*) -- The unique name for the requested shared memory, specified as a string. When creating a new shared memory block, if `None` (the default) is supplied for the name, a novel name will be generated.

— **create** (bool) -- Control whether a new shared memory block is created (True) or an existing shared memory block is attached (False).
— **size** (int) -- The requested number of bytes when creating a new shared memory block. Because some platforms choose to allocate chunks of memory based upon that platform's memory page size, the exact size of the shared memory block may be larger or equal to the size requested. When attaching to an existing shared memory block, the *size* parameter is ignored.
— **track** (bool) -- When True, register the shared memory block with a resource tracker process on platforms where the OS does not do this automatically. The resource tracker ensures proper cleanup of the shared memory even if all other processes with access to the memory exit without doing so. Python processes created from a common ancestor using *multiprocessing* facilities share a single resource tracker process, and the lifetime of shared memory segments is handled automatically among these processes. Python processes created in any other way will receive their own resource tracker when accessing shared memory with *track* enabled. This will cause the shared memory to be deleted by the resource tracker of the first process that terminates. To avoid this issue, users of *subprocess* or standalone Python processes should set *track* to False when there is already another process in place that does the bookkeeping. *track* is ignored on Windows, which has its own tracking and automatically deletes shared memory when all handles to it have been closed.

Modifié dans la version 3.13 : Added the *track* parameter.

close()

Close the file descriptor/handle to the shared memory from this instance. *close()* should be called once access to the shared memory block from this instance is no longer needed. Depending on operating system, the underlying memory may or may not be freed even if all handles to it have been closed. To ensure proper cleanup, use the *unlink()* method.

unlink()

Delete the underlying shared memory block. This should be called only once per shared memory block regardless of the number of handles to it, even in other processes. *unlink()* and *close()* can be called in any order, but trying to access data inside a shared memory block after *unlink()* may result in memory access errors, depending on platform.

This method has no effect on Windows, where the only way to delete a shared memory block is to close all handles.

buf

Une *memoryview* du contenu du bloc de mémoire partagée.

name

Nom unique du bloc de mémoire partagée (lecture seule).

size

Taille en octets du bloc de mémoire partagée (lecture seule).

L'exemple qui suit montre un exemple d'utilisation bas niveau d'instances de *SharedMemory* :

```
>>> from multiprocessing import shared_memory
>>> shm_a = shared_memory.SharedMemory(create=True, size=10)
>>> type(shm_a.buf)
<class 'memoryview'>
>>> buffer = shm_a.buf
>>> len(buffer)
10
>>> buffer[:4] = bytearray([22, 33, 44, 55])  # Modify multiple at once
>>> buffer[4] = 100                           # Modify single byte at a time
>>> # Attach to an existing shared memory block
>>> shm_b = shared_memory.SharedMemory(shm_a.name)
>>> import array
>>> array.array('b', shm_b.buf[:5])  # Copy the data into a new array.array
```

(suite sur la page suivante)

(suite de la page précédente)

```
array('b', [22, 33, 44, 55, 100])
>>> shm_b.buf[:5] = b'howdy'  # Modify via shm_b using bytes
>>> bytes(shm_a.buf[:5])      # Access via shm_a
b'howdy'
>>> shm_b.close()   # Close each SharedMemory instance
>>> shm_a.close()
>>> shm_a.unlink()  # Call unlink only once to release the shared memory
```

The following example demonstrates a practical use of the `SharedMemory` class with NumPy arrays, accessing the same `numpy.ndarray` from two distinct Python shells :

```
>>> # In the first Python interactive shell
>>> import numpy as np
>>> a = np.array([1, 1, 2, 3, 5, 8])  # Start with an existing NumPy array
>>> from multiprocessing import shared_memory
>>> shm = shared_memory.SharedMemory(create=True, size=a.nbytes)
>>> # Now create a NumPy array backed by shared memory
>>> b = np.ndarray(a.shape, dtype=a.dtype, buffer=shm.buf)
>>> b[:] = a[:]  # Copy the original data into shared memory
>>> b
array([1, 1, 2, 3, 5, 8])
>>> type(b)
<class 'numpy.ndarray'>
>>> type(a)
<class 'numpy.ndarray'>
>>> shm.name  # We did not specify a name so one was chosen for us
'psm_21467_46075'

>>> # In either the same shell or a new Python shell on the same machine
>>> import numpy as np
>>> from multiprocessing import shared_memory
>>> # Attach to the existing shared memory block
>>> existing_shm = shared_memory.SharedMemory(name='psm_21467_46075')
>>> # Note that a.shape is (6,) and a.dtype is np.int64 in this example
>>> c = np.ndarray((6,), dtype=np.int64, buffer=existing_shm.buf)
>>> c
array([1, 1, 2, 3, 5, 8])
>>> c[-1] = 888
>>> c
array([  1,   1,   2,   3,   5, 888])

>>> # Back in the first Python interactive shell, b reflects this change
>>> b
array([  1,   1,   2,   3,   5, 888])

>>> # Clean up from within the second Python shell
>>> del c  # Unnecessary; merely emphasizing the array is no longer used
>>> existing_shm.close()

>>> # Clean up from within the first Python shell
>>> del b  # Unnecessary; merely emphasizing the array is no longer used
>>> shm.close()
>>> shm.unlink()  # Free and release the shared memory block at the very end
```

class multiprocessing.managers.**SharedMemoryManager**([*address*[, *authkey*]])

A subclass of `multiprocessing.managers.BaseManager` which can be used for the management of shared memory blocks across processes.

A call to *start()* on a SharedMemoryManager instance causes a new process to be started. This new process's sole purpose is to manage the life cycle of all shared memory blocks created through it. To trigger the release of all shared memory blocks managed by that process, call *shutdown()* on the instance. This triggers a *unlink()* call on all of the *SharedMemory* objects managed by that process and then stops the process itself. By creating SharedMemory instances through a SharedMemoryManager, we avoid the need to manually track and trigger the freeing of shared memory resources.

Cette classe fournit des méthodes pour créer et renvoyer des instances de *SharedMemory* et pour créer des objets compatibles liste (*ShareableList*) basés sur la mémoire partagée.

Refer to *BaseManager* for a description of the inherited *address* and *authkey* optional input arguments and how they may be used to connect to an existing SharedMemoryManager service from other processes.

SharedMemory(*size*)

Create and return a new *SharedMemory* object with the specified *size* in bytes.

ShareableList(*sequence*)

Create and return a new *ShareableList* object, initialized by the values from the input *sequence*.

The following example demonstrates the basic mechanisms of a *SharedMemoryManager* :

```
>>> from multiprocessing.managers import SharedMemoryManager
>>> smm = SharedMemoryManager()
>>> smm.start()  # Start the process that manages the shared memory blocks
>>> sl = smm.ShareableList(range(4))
>>> sl
ShareableList([0, 1, 2, 3], name='psm_6572_7512')
>>> raw_shm = smm.SharedMemory(size=128)
>>> another_sl = smm.ShareableList('alpha')
>>> another_sl
ShareableList(['a', 'l', 'p', 'h', 'a'], name='psm_6572_12221')
>>> smm.shutdown()  # Calls unlink() on sl, raw_shm, and another_sl
```

The following example depicts a potentially more convenient pattern for using *SharedMemoryManager* objects via the with statement to ensure that all shared memory blocks are released after they are no longer needed :

```
>>> with SharedMemoryManager() as smm:
...     sl = smm.ShareableList(range(2000))
...     # Divide the work among two processes, storing partial results in sl
...     p1 = Process(target=do_work, args=(sl, 0, 1000))
...     p2 = Process(target=do_work, args=(sl, 1000, 2000))
...     p1.start()
...     p2.start()  # A multiprocessing.Pool might be more efficient
...     p1.join()
...     p2.join()   # Wait for all work to complete in both processes
...     total_result = sum(sl)  # Consolidate the partial results now in sl
```

When using a *SharedMemoryManager* in a with statement, the shared memory blocks created using that manager are all released when the with statement's code block finishes execution.

class multiprocessing.shared_memory.**ShareableList**(*sequence=None*, *, *name=None*)

Provide a mutable list-like object where all values stored within are stored in a shared memory block. This constrains storable values to the following built-in data types :

— *int* (signed 64-bit)
— *float*
— *bool*
— *str* (less than 10M bytes each when encoded as UTF-8)
— *bytes* (less than 10M bytes each)
— None

It also notably differs from the built-in *list* type in that these lists can not change their overall length (i.e. no append(), insert(), etc.) and do not support the dynamic creation of new ShareableList instances via

slicing.

sequence is used in populating a new `ShareableList` full of values. Set to `None` to instead attach to an already existing `ShareableList` by its unique shared memory name.

name is the unique name for the requested shared memory, as described in the definition for `SharedMemory`. When attaching to an existing `ShareableList`, specify its shared memory block's unique name while leaving *sequence* set to `None`.

> **Note**
>
> A known issue exists for `bytes` and `str` values. If they end with `\x00` nul bytes or characters, those may be *silently stripped* when fetching them by index from the `ShareableList`. This `.rstrip(b'\x00')` behavior is considered a bug and may go away in the future. See gh-106939.

For applications where rstripping of trailing nulls is a problem, work around it by always unconditionally appending an extra non-0 byte to the end of such values when storing and unconditionally removing it when fetching :

```
>>> from multiprocessing import shared_memory
>>> nul_bug_demo = shared_memory.ShareableList(['?\x00', b'\x03\x02\x01\x00\
↪x00\x00'])
>>> nul_bug_demo[0]
'?'
>>> nul_bug_demo[1]
b'\x03\x02\x01'
>>> nul_bug_demo.shm.unlink()
>>> padded = shared_memory.ShareableList(['?\x00\x07', b'\x03\x02\x01\x00\x00\
↪x00\x07'])
>>> padded[0][:-1]
'?\x00'
>>> padded[1][:-1]
b'\x03\x02\x01\x00\x00\x00'
>>> padded.shm.unlink()
```

count(*value*)

Return the number of occurrences of *value*.

index(*value*)

Return first index position of *value*. Raise `ValueError` if *value* is not present.

format

Attribut en lecture seule contenant le format d'agrégation `struct` utilisé par les valeurs déjà stockées.

shm

Instance de `SharedMemory` dans laquelle les valeurs sont stockées.

L'exemple qui suit illustre un cas d'usage de base d'une instance de `ShareableList` :

```
>>> from multiprocessing import shared_memory
>>> a = shared_memory.ShareableList(['howdy', b'HoWdY', -273.154, 100, None, True,␣
↪42])
>>> [ type(entry) for entry in a ]
[<class 'str'>, <class 'bytes'>, <class 'float'>, <class 'int'>, <class 'NoneType'>
↪, <class 'bool'>, <class 'int'>]
>>> a[2]
-273.154
>>> a[2] = -78.5
>>> a[2]
```

(suite sur la page suivante)

(suite de la page précédente)

```
-78.5
>>> a[2] = 'dry ice'  # Changing data types is supported as well
>>> a[2]
'dry ice'
>>> a[2] = 'larger than previously allocated storage space'
Traceback (most recent call last):
  ...
ValueError: exceeds available storage for existing str
>>> a[2]
'dry ice'
>>> len(a)
7
>>> a.index(42)
6
>>> a.count(b'howdy')
0
>>> a.count(b'HoWdY')
1
>>> a.shm.close()
>>> a.shm.unlink()
>>> del a  # Use of a ShareableList after call to unlink() is unsupported
```

L'exemple ci-dessous montre comment un, deux ou un grand nombre de processus peuvent accéder à une `ShareableList` commune à partir du nom du bloc mémoire partagé sous-jacent :

```
>>> b = shared_memory.ShareableList(range(5))          # In a first process
>>> c = shared_memory.ShareableList(name=b.shm.name)   # In a second process
>>> c
ShareableList([0, 1, 2, 3, 4], name='...')
>>> c[-1] = -999
>>> b[-1]
-999
>>> b.shm.close()
>>> c.shm.close()
>>> c.shm.unlink()
```

The following examples demonstrates that `ShareableList` (and underlying `SharedMemory`) objects can be pickled and unpickled if needed. Note, that it will still be the same shared object. This happens, because the deserialized object has the same unique name and is just attached to an existing object with the same name (if the object is still alive) :

```
>>> import pickle
>>> from multiprocessing import shared_memory
>>> sl = shared_memory.ShareableList(range(10))
>>> list(sl)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> deserialized_sl = pickle.loads(pickle.dumps(sl))
>>> list(deserialized_sl)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> sl[0] = -1
>>> deserialized_sl[1] = -2
>>> list(sl)
[-1, -2, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(deserialized_sl)
[-1, -2, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> sl.shm.close()
>>> sl.shm.unlink()
```

18.4 The concurrent package

Il n'y a actuellement qu'un module dans ce paquet :

— *concurrent.futures* -- Lancer des tâches en parallèle

18.5 concurrent.futures --- Launching parallel tasks

Ajouté dans la version 3.2.

Source code : Lib/concurrent/futures/thread.py and Lib/concurrent/futures/process.py

---

The *concurrent.futures* module provides a high-level interface for asynchronously executing callables.

The asynchronous execution can be performed with threads, using *ThreadPoolExecutor* or *InterpreterPoolExecutor*, or separate processes, using *ProcessPoolExecutor*. Each implements the same interface, which is defined by the abstract *Executor* class.

Availability : not WASI.

This module does not work or is not available on WebAssembly. See *Plateformes WebAssembly* for more information.

18.5.1 Executor Objects

class concurrent.futures.**Executor**

An abstract class that provides methods to execute calls asynchronously. It should not be used directly, but through its concrete subclasses.

submit(*fn*, /, **args*, ***kwargs*)

Schedules the callable, *fn*, to be executed as `fn(*args, **kwargs)` and returns a *Future* object representing the execution of the callable.

```
with ThreadPoolExecutor(max_workers=1) as executor:
    future = executor.submit(pow, 323, 1235)
    print(future.result())
```

map(*fn*, **iterables*, *timeout=None*, *chunksize=1*)

Similar to *map(fn, *iterables)* except :

— the *iterables* are collected immediately rather than lazily ;

— *fn* is executed asynchronously and several calls to *fn* may be made concurrently.

The returned iterator raises a *TimeoutError* if *__next__()* is called and the result isn't available after *timeout* seconds from the original call to *Executor.map()*. *timeout* can be an int or a float. If *timeout* is not specified or `None`, there is no limit to the wait time.

If a *fn* call raises an exception, then that exception will be raised when its value is retrieved from the iterator.

When using *ProcessPoolExecutor*, this method chops *iterables* into a number of chunks which it submits to the pool as separate tasks. The (approximate) size of these chunks can be specified by setting *chunksize* to a positive integer. For very long iterables, using a large value for *chunksize* can significantly improve performance compared to the default size of 1. With *ThreadPoolExecutor* and *InterpreterPoolExecutor*, *chunksize* has no effect.

Modifié dans la version 3.5 : Added the *chunksize* argument.

shutdown(*wait=True*, *, *cancel_futures=False*)

Signal the executor that it should free any resources that it is using when the currently pending futures are done executing. Calls to `Executor.submit()` and `Executor.map()` made after shutdown will raise `RuntimeError`.

If *wait* is `True` then this method will not return until all the pending futures are done executing and the resources associated with the executor have been freed. If *wait* is `False` then this method will return immediately and the resources associated with the executor will be freed when all pending futures are done executing. Regardless of the value of *wait*, the entire Python program will not exit until all pending futures are done executing.

If *cancel_futures* is `True`, this method will cancel all pending futures that the executor has not started running. Any futures that are completed or running won't be cancelled, regardless of the value of *cancel_futures*.

If both *cancel_futures* and *wait* are `True`, all futures that the executor has started running will be completed prior to this method returning. The remaining futures are cancelled.

You can avoid having to call this method explicitly if you use the `with` statement, which will shutdown the `Executor` (waiting as if `Executor.shutdown()` were called with *wait* set to `True`) :

```
import shutil
with ThreadPoolExecutor(max_workers=4) as e:
    e.submit(shutil.copy, 'src1.txt', 'dest1.txt')
    e.submit(shutil.copy, 'src2.txt', 'dest2.txt')
    e.submit(shutil.copy, 'src3.txt', 'dest3.txt')
    e.submit(shutil.copy, 'src4.txt', 'dest4.txt')
```

Modifié dans la version 3.9 : Added *cancel_futures*.

18.5.2 ThreadPoolExecutor

`ThreadPoolExecutor` is an `Executor` subclass that uses a pool of threads to execute calls asynchronously.

Deadlocks can occur when the callable associated with a `Future` waits on the results of another `Future`. For example :

```
import time
def wait_on_b():
    time.sleep(5)
    print(b.result())  # b will never complete because it is waiting on a.
    return 5

def wait_on_a():
    time.sleep(5)
    print(a.result())  # a will never complete because it is waiting on b.
    return 6


executor = ThreadPoolExecutor(max_workers=2)
a = executor.submit(wait_on_b)
b = executor.submit(wait_on_a)
```

Et :

```
def wait_on_future():
    f = executor.submit(pow, 5, 2)
    # This will never complete because there is only one worker thread and
    # it is executing this function.
    print(f.result())
```

(suite sur la page suivante)

(suite de la page précédente)

```
executor = ThreadPoolExecutor(max_workers=1)
executor.submit(wait_on_future)
```

class concurrent.futures.**ThreadPoolExecutor**(*max_workers=None*, *thread_name_prefix=''*, *initializer=None*, *initargs=()*)

An `Executor` subclass that uses a pool of at most *max_workers* threads to execute calls asynchronously.

All threads enqueued to `ThreadPoolExecutor` will be joined before the interpreter can exit. Note that the exit handler which does this is executed *before* any exit handlers added using `atexit`. This means exceptions in the main thread must be caught and handled in order to signal threads to exit gracefully. For this reason, it is recommended that `ThreadPoolExecutor` not be used for long-running tasks.

initializer is an optional callable that is called at the start of each worker thread ; *initargs* is a tuple of arguments passed to the initializer. Should *initializer* raise an exception, all currently pending jobs will raise a `BrokenThreadPool`, as well as any attempt to submit more jobs to the pool.

Modifié dans la version 3.5 : If *max_workers* is `None` or not given, it will default to the number of processors on the machine, multiplied by `5`, assuming that `ThreadPoolExecutor` is often used to overlap I/O instead of CPU work and the number of workers should be higher than the number of workers for `ProcessPoolExecutor`.

Modifié dans la version 3.6 : Added the *thread_name_prefix* parameter to allow users to control the `threading.Thread` names for worker threads created by the pool for easier debugging.

Modifié dans la version 3.7 : Added the *initializer* and *initargs* arguments.

Modifié dans la version 3.8 : Default value of *max_workers* is changed to `min(32, os.cpu_count() + 4)`. This default value preserves at least 5 workers for I/O bound tasks. It utilizes at most 32 CPU cores for CPU bound tasks which release the GIL. And it avoids using very large resources implicitly on many-core machines.

ThreadPoolExecutor now reuses idle worker threads before starting *max_workers* worker threads too.

Modifié dans la version 3.13 : Default value of *max_workers* is changed to `min(32, (os.process_cpu_count() or 1) + 4)`.

ThreadPoolExecutor Example

```
import concurrent.futures
import urllib.request

URLS = ['http://www.foxnews.com/',
        'http://www.cnn.com/',
        'http://europe.wsj.com/',
        'http://www.bbc.co.uk/',
        'http://nonexistent-subdomain.python.org/']

# Retrieve a single page and report the URL and contents
def load_url(url, timeout):
    with urllib.request.urlopen(url, timeout=timeout) as conn:
        return conn.read()

# We can use a with statement to ensure threads are cleaned up promptly
with concurrent.futures.ThreadPoolExecutor(max_workers=5) as executor:
    # Start the load operations and mark each future with its URL
    future_to_url = {executor.submit(load_url, url, 60): url for url in URLS}
    for future in concurrent.futures.as_completed(future_to_url):
        url = future_to_url[future]
        try:
```

(suite sur la page suivante)

(suite de la page précédente)

```
            data = future.result()
        except Exception as exc:
            print('%r generated an exception: %s' % (url, exc))
        else:
            print('%r page is %d bytes' % (url, len(data)))
```

18.5.3 InterpreterPoolExecutor

The `InterpreterPoolExecutor` class uses a pool of interpreters to execute calls asynchronously. It is a `ThreadPoolExecutor` subclass, which means each worker is running in its own thread. The difference here is that each worker has its own interpreter, and runs each task using that interpreter.

The biggest benefit to using interpreters instead of only threads is true multi-core parallelism. Each interpreter has its own *Global Interpreter Lock*, so code running in one interpreter can run on one CPU core, while code in another interpreter runs unblocked on a different core.

The tradeoff is that writing concurrent code for use with multiple interpreters can take extra effort. However, this is because it forces you to be deliberate about how and when interpreters interact, and to be explicit about what data is shared between interpreters. This results in several benefits that help balance the extra effort, including true multi-core parallelism, For example, code written this way can make it easier to reason about concurrency. Another major benefit is that you don't have to deal with several of the big pain points of using threads, like race conditions.

Each worker's interpreter is isolated from all the other interpreters. "Isolated" means each interpreter has its own runtime state and operates completely independently. For example, if you redirect `sys.stdout` in one interpreter, it will not be automatically redirected any other interpreter. If you import a module in one interpreter, it is not automatically imported in any other. You would need to import the module separately in interpreter where you need it. In fact, each module imported in an interpreter is a completely separate object from the same module in a different interpreter, including `sys`, `builtins`, and even `__main__`.

Isolation means a mutable object, or other data, cannot be used by more than one interpreter at the same time. That effectively means interpreters cannot actually share such objects or data. Instead, each interpreter must have its own copy, and you will have to synchronize any changes between the copies manually. Immutable objects and data, like the builtin singletons, strings, and tuples of immutable objects, don't have these limitations.

Communicating and synchronizing between interpreters is most effectively done using dedicated tools, like those proposed in **PEP 734**. One less efficient alternative is to serialize with `pickle` and then send the bytes over a shared `socket` or `pipe`.

class `concurrent.futures.`**`InterpreterPoolExecutor`**(*max_workers=None*, *thread_name_prefix=''*, *initializer=None*, *initargs=()*, *shared=None*)

A `ThreadPoolExecutor` subclass that executes calls asynchronously using a pool of at most *max_workers* threads. Each thread runs tasks in its own interpreter. The worker interpreters are isolated from each other, which means each has its own runtime state and that they can't share any mutable objects or other data. Each interpreter has its own *Global Interpreter Lock*, which means code run with this executor has true multi-core parallelism.

The optional *initializer* and *initargs* arguments have the same meaning as for `ThreadPoolExecutor` : the initializer is run when each worker is created, though in this case it is run.in the worker's interpreter. The executor serializes the *initializer* and *initargs* using `pickle` when sending them to the worker's interpreter.

> **Note**
>
> Functions defined in the `__main__` module cannot be pickled and thus cannot be used.

> **Note**
>
> The executor may replace uncaught exceptions from *initializer* with `ExecutionFailed`.

The optional *shared* argument is a `dict` of objects that all interpreters in the pool share. The *shared* items are added to each interpreter's `__main__` module. Not all objects are shareable. Shareable objects include the builtin singletons, `str` and `bytes`, and `memoryview`. See **PEP 734** for more info.

Other caveats from parent `ThreadPoolExecutor` apply here.

`submit()` and `map()` work like normal, except the worker serializes the callable and arguments using `pickle` when sending them to its interpreter. The worker likewise serializes the return value when sending it back.

> **Note**
>
> Functions defined in the `__main__` module cannot be pickled and thus cannot be used.

When a worker's current task raises an uncaught exception, the worker always tries to preserve the exception as-is. If that is successful then it also sets the `__cause__` to a corresponding `ExecutionFailed` instance, which contains a summary of the original exception. In the uncommon case that the worker is not able to preserve the original as-is then it directly preserves the corresponding `ExecutionFailed` instance instead.

18.5.4 ProcessPoolExecutor

The `ProcessPoolExecutor` class is an `Executor` subclass that uses a pool of processes to execute calls asynchronously. `ProcessPoolExecutor` uses the `multiprocessing` module, which allows it to side-step the *Global Interpreter Lock* but also means that only picklable objects can be executed and returned.

The `__main__` module must be importable by worker subprocesses. This means that `ProcessPoolExecutor` will not work in the interactive interpreter.

Calling `Executor` or `Future` methods from a callable submitted to a `ProcessPoolExecutor` will result in deadlock.

class `concurrent.futures.`**`ProcessPoolExecutor`**(*max_workers=None*, *mp_context=None*, *initializer=None*, *initargs=()*, *max_tasks_per_child=None*)

An `Executor` subclass that executes calls asynchronously using a pool of at most *max_workers* processes. If *max_workers* is `None` or not given, it will default to `os.process_cpu_count()`. If *max_workers* is less than or equal to `0`, then a `ValueError` will be raised. On Windows, *max_workers* must be less than or equal to `61`. If it is not then `ValueError` will be raised. If *max_workers* is `None`, then the default chosen will be at most `61`, even if more processors are available. *mp_context* can be a `multiprocessing` context or `None`. It will be used to launch the workers. If *mp_context* is `None` or not given, the default `multiprocessing` context is used. See *Contextes et méthodes de démarrage*.

initializer is an optional callable that is called at the start of each worker process; *initargs* is a tuple of arguments passed to the initializer. Should *initializer* raise an exception, all currently pending jobs will raise a `BrokenProcessPool`, as well as any attempt to submit more jobs to the pool.

max_tasks_per_child is an optional argument that specifies the maximum number of tasks a single process can execute before it will exit and be replaced with a fresh worker process. By default *max_tasks_per_child* is `None` which means worker processes will live as long as the pool. When a max is specified, the "spawn" multiprocessing start method will be used by default in absence of a *mp_context* parameter. This feature is incompatible with the "fork" start method.

Modifié dans la version 3.3 : When one of the worker processes terminates abruptly, a `BrokenProcessPool` error is now raised. Previously, behaviour was undefined but operations on the executor or its futures would often freeze or deadlock.

Modifié dans la version 3.7 : The *mp_context* argument was added to allow users to control the start_method for worker processes created by the pool.

Added the *initializer* and *initargs* arguments.

Modifié dans la version 3.11 : The *max_tasks_per_child* argument was added to allow users to control the lifetime of workers in the pool.

Modifié dans la version 3.12 : On POSIX systems, if your application has multiple threads and the *multiprocessing* context uses the "fork" start method : The *os.fork()* function called internally to spawn workers may raise a *DeprecationWarning*. Pass a *mp_context* configured to use a different start method. See the *os.fork()* documentation for further explanation.

Modifié dans la version 3.13 : *max_workers* uses *os.process_cpu_count()* by default, instead of *os.cpu_count()*.

Modifié dans la version 3.14 : The default process start method (see *Contextes et méthodes de démarrage*) changed away from *fork*. If you require the *fork* start method for *ProcessPoolExecutor* you must explicitly pass mp_context=multiprocessing.get_context("fork").

ProcessPoolExecutor Example

```
import concurrent.futures
import math

PRIMES = [
    112272535095293,
    112582705942171,
    112272535095293,
    115280095190773,
    115797848077099,
    1099726899285419]

def is_prime(n):
    if n < 2:
        return False
    if n == 2:
        return True
    if n % 2 == 0:
        return False

    sqrt_n = int(math.floor(math.sqrt(n)))
    for i in range(3, sqrt_n + 1, 2):
        if n % i == 0:
            return False
    return True

def main():
    with concurrent.futures.ProcessPoolExecutor() as executor:
        for number, prime in zip(PRIMES, executor.map(is_prime, PRIMES)):
            print('%d is prime: %s' % (number, prime))

if __name__ == '__main__':
    main()
```

18.5.5 Future Objects

The *Future* class encapsulates the asynchronous execution of a callable. *Future* instances are created by *Executor.submit()*.

class concurrent.futures.**Future**

Encapsulates the asynchronous execution of a callable. *Future* instances are created by *Executor.submit()* and should not be created directly except for testing.

cancel()

Attempt to cancel the call. If the call is currently being executed or finished running and cannot be cancelled then the method will return False, otherwise the call will be cancelled and the method will

return `True`.

cancelled()

Return `True` if the call was successfully cancelled.

running()

Return `True` if the call is currently being executed and cannot be cancelled.

done()

Return `True` if the call was successfully cancelled or finished running.

result(*timeout=None*)

Return the value returned by the call. If the call hasn't yet completed then this method will wait up to *timeout* seconds. If the call hasn't completed in *timeout* seconds, then a `TimeoutError` will be raised. *timeout* can be an int or float. If *timeout* is not specified or `None`, there is no limit to the wait time.

If the future is cancelled before completing then `CancelledError` will be raised.

If the call raised an exception, this method will raise the same exception.

exception(*timeout=None*)

Return the exception raised by the call. If the call hasn't yet completed then this method will wait up to *timeout* seconds. If the call hasn't completed in *timeout* seconds, then a `TimeoutError` will be raised. *timeout* can be an int or float. If *timeout* is not specified or `None`, there is no limit to the wait time.

If the future is cancelled before completing then `CancelledError` will be raised.

If the call completed without raising, `None` is returned.

add_done_callback(*fn*)

Attaches the callable *fn* to the future. *fn* will be called, with the future as its only argument, when the future is cancelled or finishes running.

Added callables are called in the order that they were added and are always called in a thread belonging to the process that added them. If the callable raises an `Exception` subclass, it will be logged and ignored. If the callable raises a `BaseException` subclass, the behavior is undefined.

If the future has already completed or been cancelled, *fn* will be called immediately.

The following `Future` methods are meant for use in unit tests and `Executor` implementations.

set_running_or_notify_cancel()

This method should only be called by `Executor` implementations before executing the work associated with the `Future` and by unit tests.

If the method returns `False` then the `Future` was cancelled, i.e. `Future.cancel()` was called and returned `True`. Any threads waiting on the `Future` completing (i.e. through `as_completed()` or `wait()`) will be woken up.

If the method returns `True` then the `Future` was not cancelled and has been put in the running state, i.e. calls to `Future.running()` will return `True`.

This method can only be called once and cannot be called after `Future.set_result()` or `Future.set_exception()` have been called.

set_result(*result*)

Sets the result of the work associated with the `Future` to *result*.

This method should only be used by `Executor` implementations and unit tests.

Modifié dans la version 3.8 : This method raises `concurrent.futures.InvalidStateError` if the `Future` is already done.

set_exception(*exception*)

Sets the result of the work associated with the `Future` to the `Exception` *exception*.

This method should only be used by `Executor` implementations and unit tests.

Modifié dans la version 3.8 : This method raises `concurrent.futures.InvalidStateError` if the `Future` is already done.

18.5.6 Module Functions

`concurrent.futures.wait`(*fs*, *timeout=None*, *return_when=ALL_COMPLETED*)

Wait for the `Future` instances (possibly created by different `Executor` instances) given by *fs* to complete. Duplicate futures given to *fs* are removed and will be returned only once. Returns a named 2-tuple of sets. The first set, named `done`, contains the futures that completed (finished or cancelled futures) before the wait completed. The second set, named `not_done`, contains the futures that did not complete (pending or running futures).

timeout can be used to control the maximum number of seconds to wait before returning. *timeout* can be an int or float. If *timeout* is not specified or `None`, there is no limit to the wait time.

return_when indique quand la fonction doit se terminer. Il peut prendre les valeurs suivantes :

Constante	Description
`concurrent.futures.FIRST_COMPLETED`	La fonction se termine lorsque n'importe quel futur se termine ou est annulé.
`concurrent.futures.FIRST_EXCEPTION`	The function will return when any future finishes by raising an exception. If no future raises an exception then it is equivalent to `ALL_COMPLETED`.
`concurrent.futures.ALL_COMPLETED`	La fonction se termine lorsque les *futurs* sont tous finis ou annulés.

`concurrent.futures.as_completed`(*fs*, *timeout=None*)

Returns an iterator over the `Future` instances (possibly created by different `Executor` instances) given by *fs* that yields futures as they complete (finished or cancelled futures). Any futures given by *fs* that are duplicated will be returned once. Any futures that completed before `as_completed()` is called will be yielded first. The returned iterator raises a `TimeoutError` if `__next__()` is called and the result isn't available after *timeout* seconds from the original call to `as_completed()`. *timeout* can be an int or float. If *timeout* is not specified or `None`, there is no limit to the wait time.

Voir aussi

PEP 3148 -- futures - execute computations asynchronously
The proposal which described this feature for inclusion in the Python standard library.

18.5.7 Exception classes

exception `concurrent.futures.CancelledError`

Raised when a future is cancelled.

exception `concurrent.futures.TimeoutError`

A deprecated alias of `TimeoutError`, raised when a future operation exceeds the given timeout.

Modifié dans la version 3.11 : cette classe est devenue un alias de `TimeoutError`.

exception `concurrent.futures.BrokenExecutor`

Derived from `RuntimeError`, this exception class is raised when an executor is broken for some reason, and cannot be used to submit or execute new tasks.

Ajouté dans la version 3.7.

exception concurrent.futures.**InvalidStateError**

Raised when an operation is performed on a future that is not allowed in the current state.

Ajouté dans la version 3.8.

exception concurrent.futures.thread.**BrokenThreadPool**

Derived from *BrokenExecutor*, this exception class is raised when one of the workers of a *ThreadPoolExecutor* has failed initializing.

Ajouté dans la version 3.7.

exception concurrent.futures.interpreter.**BrokenInterpreterPool**

Derived from *BrokenThreadPool*, this exception class is raised when one of the workers of a *InterpreterPoolExecutor* has failed initializing.

Ajouté dans la version 3.14.

exception concurrent.futures.interpreter.**ExecutionFailed**

Raised from *InterpreterPoolExecutor* when the given initializer fails or from *submit()* when there's an uncaught exception from the submitted task.

Ajouté dans la version 3.14.

exception concurrent.futures.process.**BrokenProcessPool**

Derived from *BrokenExecutor* (formerly *RuntimeError*), this exception class is raised when one of the workers of a *ProcessPoolExecutor* has terminated in a non-clean fashion (for example, if it was killed from the outside).

Ajouté dans la version 3.3.

18.6 subprocess --- Subprocess management

Code source : Lib/subprocess.py

Le module *subprocess* vous permet de lancer de nouveaux processus, les connecter à des tubes d'entrée/sortie/erreur, et d'obtenir leurs codes de retour. Ce module a l'intention de remplacer plusieurs anciens modules et fonctions :

```
os.system
os.spawn*
```

De plus amples informations sur comment le module *subprocess* peut être utilisé pour remplacer ces modules et fonctions peuvent être trouvées dans les sections suivantes.

Voir aussi

PEP 324 -- PEP proposant le module *subprocess*

Availability : not Android, not iOS, not WASI.

This module is not supported on *mobile platforms* or *WebAssembly platforms*.

18.6.1 Utiliser le module subprocess

L'approche recommandée pour invoquer un sous-processus est d'utiliser la fonction *run()* pour tous les cas d'utilisation qu'elle gère. Pour les cas d'utilisation plus avancés, l'interface inhérente *Popen* peut être utilisée directement.

subprocess.**run**(*args*, *, *stdin=None*, *input=None*, *stdout=None*, *stderr=None*, *capture_output=False*, *shell=False*, *cwd=None*, *timeout=None*, *check=False*, *encoding=None*, *errors=None*, *text=None*, *env=None*, *universal_newlines=None*, **other_popen_kwargs*)

Lance la commande décrite par *args*. Attend que la commande se termine, puis renvoie une instance `CompletedProcess`.

Les arguments présentés ci-dessus sont simplement les plus utilisés, décrits ci-dessous dans la section *Arguments fréquemment utilisés* (d'où l'utilisation de la notation *keyword-only* dans la signature abrégée). La signature complète de la fonction est sensiblement la même que celle du constructeur de `Popen`, à l'exception de *timeout*, *input*, *check* et *capture_output*, tous les arguments donnés à cette fonction passent à travers cette interface.

If *capture_output* is true, stdout and stderr will be captured. When used, the internal `Popen` object is automatically created with *stdout* and *stderr* both set to `PIPE`. The *stdout* and *stderr* arguments may not be supplied at the same time as *capture_output*. If you wish to capture and combine both streams into one, set *stdout* to `PIPE` and *stderr* to `STDOUT`, instead of using *capture_output*.

A *timeout* may be specified in seconds, it is internally passed on to `Popen.communicate()`. If the timeout expires, the child process will be killed and waited for. The `TimeoutExpired` exception will be re-raised after the child process has terminated. The initial process creation itself cannot be interrupted on many platform APIs so you are not guaranteed to see a timeout exception until at least after however long process creation takes.

The *input* argument is passed to `Popen.communicate()` and thus to the subprocess's stdin. If used it must be a byte sequence, or a string if *encoding* or *errors* is specified or *text* is true. When used, the internal `Popen` object is automatically created with *stdin* set to `PIPE`, and the *stdin* argument may not be used as well.

Si *check* est vrai, et que le processus s'arrête avec un code de statut non nul, une exception `CalledProcessError` est levée. Les attributs de cette exception contiennent les arguments, le code de statut, et les sorties standard et d'erreur si elles ont été capturées.

Si *encoding* ou *errors* sont spécifiés, ou *text* est vrai, les fichiers pour les entrées et sorties sont ouverts en mode texte en utilisant les paramètres *encoding* et *errors* spécifiés, ou les valeurs par défaut de `io.TextIOWrapper`. L'argument *universal_newlines* est équivalent à *text* et est fourni pour la rétrocompatibilité. Par défaut, les fichiers sont ouverts en mode binaire.

If *env* is not `None`, it must be a mapping that defines the environment variables for the new process; these are used instead of the default behavior of inheriting the current process' environment. It is passed directly to `Popen`. This mapping can be str to str on any platform or bytes to bytes on POSIX platforms much like `os.environ` or `os.environb`.

Exemples :

```
>>> subprocess.run(["ls", "-l"])  # doesn't capture output
CompletedProcess(args=['ls', '-l'], returncode=0)

>>> subprocess.run("exit 1", shell=True, check=True)
Traceback (most recent call last):
  ...
subprocess.CalledProcessError: Command 'exit 1' returned non-zero exit status 1

>>> subprocess.run(["ls", "-l", "/dev/null"], capture_output=True)
CompletedProcess(args=['ls', '-l', '/dev/null'], returncode=0,
stdout=b'crw-rw-rw- 1 root root 1, 3 Jan 23 16:23 /dev/null\n', stderr=b'')
```

Ajouté dans la version 3.5.

Modifié dans la version 3.6 : Ajout des paramètres *encoding* et *errors*

Modifié dans la version 3.7 : Ajout du paramètre *text*, qui agit comme un alias plus compréhensible de *universal_newlines*. Ajout du paramètre *capture_output*.

Modifié dans la version 3.12 : Changed Windows shell search order for `shell=True`. The current directory and `%PATH%` are replaced with `%COMSPEC%` and `%SystemRoot%\System32\cmd.exe`. As a result, dropping

a malicious program named `cmd.exe` into a current directory no longer works.

class subprocess.**CompletedProcess**

La valeur de retour de *run()*, représentant un processus qui s'est terminé.

args

Les arguments utilisés pour lancer le processus. Cela peut être une liste ou une chaîne de caractères.

returncode

Le code de statut du processus fils. Typiquement, un code de statut de 0 indique qu'il s'est exécuté avec succès.

Une valeur négative `-N` indique que le processus enfant a été terminé par un signal `N` (seulement sur les systèmes *POSIX*).

stdout

La sortie standard capturée du processus enfant. Une séquence de *bytes*, ou une chaîne de caractères si *run()* a été appelée avec *encoding*, *errors* ou *text=True*. Vaut `None` si la sortie standard n'était pas capturée.

Si vous avez lancé le processus avec `stderr=subprocess.STDOUT`, les sorties standard et d'erreur seront combinées dans cet attribut, et *stderr* sera mis à `None`.

stderr

La sortie d'erreur capturée du processus enfant. Une séquence de *bytes*, ou une chaîne de caractères si *run()* a été appelée avec *encoding*, *errors* ou *text=True*. Vaut `None` si la sortie d'erreur n'était pas capturée.

check_returncode()

Si *returncode* n'est pas nul, lève une *CalledProcessError*.

Ajouté dans la version 3.5.

subprocess.**DEVNULL**

Valeur spéciale qui peut être utilisée pour les arguments *stdin*, *stdout* ou *stderr* de *Popen* et qui indique que le fichier spécial *os.devnull* sera utilisé.

Ajouté dans la version 3.3.

subprocess.**PIPE**

Valeur spéciale qui peut être utilisée pour les arguments *stdin*, *stdout* ou *stderr* de *Popen* et qui indique qu'un tube vers le flux standard doit être ouvert. Surtout utile avec *Popen.communicate()*.

subprocess.**STDOUT**

Valeur spéciale qui peut être utilisée pour l'argument *stderr* de *Popen* et qui indique que la sortie d'erreur doit être redirigée vers le même descripteur que la sortie standard.

exception subprocess.**SubprocessError**

Classe de base à toutes les autres exceptions du module.

Ajouté dans la version 3.3.

exception subprocess.**TimeoutExpired**

Sous-classe de *SubprocessError*, levée quand un *timeout* expire pendant l'attente d'un processus enfant.

cmd

La commande utilisée pour instancier le processus fils.

timeout

Le *timeout* en secondes.

output

Sortie du processus enfant si elle a été récupérée par *run()* ou *check_output()*. Sinon, `None`. La sortie est toujours de type *bytes* quand elle est récupérée, que le paramètre `text` soit mis à `True` ou non. Si la sortie est vide, cette dernière peut être `None` au lieu de `b''`.

stdout

Alias pour *output*, afin d'avoir une symétrie avec `stderr`.

stderr

Stderr output of the child process if it was captured by `run()`. Otherwise, `None`. This is always `bytes` when stderr output was captured regardless of the `text=True` setting. It may remain `None` instead of `b''` when no stderr output was observed.

Ajouté dans la version 3.3.

Modifié dans la version 3.5 : Ajout des attributs *stdout* et *stderr*

exception subprocess.**CalledProcessError**

Subclass of `SubprocessError`, raised when a process run by `check_call()`, `check_output()`, or `run()` (with `check=True`) returns a non-zero exit status.

returncode

Code de statut du processus fils. Si le processus a été arrêté par un signal, le code sera négatif et correspondra à l'opposé du numéro de signal.

cmd

La commande utilisée pour instancier le processus fils.

output

La sortie du processus fils, si capturée par `run()` ou `check_output()`. Autrement, `None`.

stdout

Alias pour *output*, afin d'avoir une symétrie avec `stderr`.

stderr

La sortie d'erreur du processus fils, si capturée par `run()`. Autrement, `None`.

Modifié dans la version 3.5 : Ajout des attributs *stdout* et *stderr*

Arguments fréquemment utilisés

Pour gérer un large ensemble de cas, le constructeur de `Popen` (et les fonctions de convenance) acceptent de nombreux arguments optionnels. Pour les cas d'utilisation les plus typiques, beaucoup de ces arguments peuvent sans problème être laissés à leurs valeurs par défaut. Les arguments les plus communément nécessaires sont :

args est requis pour tous les appels et doit être une chaîne de caractères ou une séquence d'arguments du programme. Il est généralement préférable de fournir une séquence d'arguments, puisque cela permet au module de s'occuper des potentiels échappements ou guillemets autour des arguments (p. ex. pour permettre des espaces dans des noms de fichiers). Si l'argument est passé comme une simple chaîne, soit *shell* doit valoir `True` (voir ci-dessous) soit la chaîne doit simplement contenir le nom du programme à exécuter sans spécifier d'arguments supplémentaires.

stdin, *stdout* and *stderr* specify the executed program's standard input, standard output and standard error file handles, respectively. Valid values are `None`, `PIPE`, `DEVNULL`, an existing file descriptor (a positive integer), and an existing *file object* with a valid file descriptor. With the default settings of `None`, no redirection will occur. `PIPE` indicates that a new pipe to the child should be created. `DEVNULL` indicates that the special file `os.devnull` will be used. Additionally, *stderr* can be `STDOUT`, which indicates that the stderr data from the child process should be captured into the same file handle as for *stdout*.

Si *encoding* ou *errors* sont spécifiés, ou si *text* (aussi appelé *universal_newlines*) est vrai, les fichiers *stdin*, *stdout* et *stderr* seront ouverts en mode texte en utilisant les *encoding* et *errors* spécifiés à l'appel, ou les valeurs par défaut de `io.TextIOWrapper`.

Pour *stdin*, les caractères de fin de ligne `'\n'` de l'entrée seront convertis vers des séparateurs de ligne par défaut `os.linesep`. Pour *stdout* et *stderr*, toutes les fins de lignes des sorties seront converties vers `'\n'`. Pour plus d'informations, voir la documentation de la classe `io.TextIOWrapper` quand l'argument *newline* du constructeur est `None`.

Si le mode texte n'est pas utilisé, *stdin*, *stdout* et *stderr* seront ouverts comme des flux binaires. Aucune conversion d'encodage ou de fins de ligne ne sera réalisée.

Modifié dans la version 3.6 : Added the *encoding* and *errors* parameters.

Modifié dans la version 3.7 : Ajout du paramètre *text* comme alias de *universal_newlines*.

> **Note**
>
> L'attribut *newlines* des objets `Popen.stdin`, `Popen.stdout` et `Popen.stderr` ne sont pas mis à jour par la méthode `Popen.communicate()`.

Si *shell* vaut `True`, la commande spécifiée sera exécutée à travers un *shell*. Cela peut être utile si vous utilisez Python pour le contrôle de flot qu'il propose par rapport à beaucoup de *shells* système et voulez tout de même profiter des autres fonctionnalités des *shells* telles que les tubes (*pipes*), les motifs de fichiers, l'expansion des variables d'environnement, et l'expansion du ~ vers le répertoire d'accueil de l'utilisateur. Cependant, notez que Python lui-même propose l'implémentation de beaucoup de ces fonctionnalités (en particulier `glob`, `fnmatch`, `os.walk()`, `os.path.expandvars()`, `os.path.expanduser()` et `shutil`).

Modifié dans la version 3.3 : Quand *universal_newlines* vaut `True`, la classe utilise l'encodage `locale.getpreferredencoding(False)` plutôt que `locale.getpreferredencoding()`. Voir la classe `io.TextIOWrapper` pour plus d'informations sur ce changement.

> **Note**
>
> Lire la section *Security Considerations* avant d'utiliser `shell=True`.

Ces options, ainsi que toutes les autres, sont décrites plus en détails dans la documentation du constructeur de `Popen`.

Constructeur de *Popen*

La création et la gestion sous-jacentes des processus est gérée par la classe `Popen`. Elle offre beaucoup de flexibilité de façon à ce que les développeurs soient capables de gérer les cas d'utilisation les moins communs, non couverts par les fonctions de convenance.

class subprocess.**Popen**(*args*, *bufsize=-1*, *executable=None*, *stdin=None*, *stdout=None*, *stderr=None*, *preexec_fn=None*, *close_fds=True*, *shell=False*, *cwd=None*, *env=None*, *universal_newlines=None*, *startupinfo=None*, *creationflags=0*, *restore_signals=True*, *start_new_session=False*, *pass_fds=()*, *, *group=None*, *extra_groups=None*, *user=None*, *umask=-1*, *encoding=None*, *errors=None*, *text=None*, *pipesize=-1*, *process_group=None*)

Execute a child program in a new process. On POSIX, the class uses `os.execvpe()`-like behavior to execute the child program. On Windows, the class uses the Windows `CreateProcess()` function. The arguments to `Popen` are as follows.

args peut être une séquence d'arguments du programme, une chaîne seule ou un *objet chemin*. Par défaut, le programme à exécuter est le premier élément de *args* si *args* est une séquence. Si *args* est une chaîne de caractères, l'interprétation dépend de la plateforme et est décrite plus bas. Voir les arguments *shell* et *executable* pour d'autres différences avec le comportement par défaut. Sans autre indication, il est recommandé de passer *args* comme une séquence.

> **Avertissement**
>
> For maximum reliability, use a fully qualified path for the executable. To search for an unqualified name on `PATH`, use `shutil.which()`. On all platforms, passing `sys.executable` is the recommended way to launch the current Python interpreter again, and use the `-m` command-line format to launch an installed module.
>
> Resolving the path of *executable* (or the first item of *args*) is platform dependent. For POSIX, see `os.execvpe()`, and note that when resolving or searching for the executable path, *cwd* overrides the current

working directory and *env* can override the `PATH` environment variable. For Windows, see the documentation of the `lpApplicationName` and `lpCommandLine` parameters of WinAPI `CreateProcess`, and note that when resolving or searching for the executable path with `shell=False`, *cwd* does not override the current working directory and *env* cannot override the `PATH` environment variable. Using a full path avoids all of these variations.

Exemple de passage d'arguments, sous forme de séquence, à un autre programme :

```
Popen(["/usr/bin/git", "commit", "-m", "Fixes a bug."])
```

Sur les systèmes *POSIX*, si *args* est une chaîne, elle est interprétée comme le nom ou le chemin du programme à exécuter. Cependant, cela ne peut être fait que si le programme est passé sans arguments.

Note

Transformer une commande shell en une séquence de commandes peut ne pas être évident. `shlex.split()` permet d'illustrer comment séquencer une commande correctement pour *args* :

```
>>> import shlex, subprocess
>>> command_line = input()
/bin/vikings -input eggs.txt -output "spam spam.txt" -cmd "echo '$MONEY'"
>>> args = shlex.split(command_line)
>>> print(args)
['/bin/vikings', '-input', 'eggs.txt', '-output', 'spam spam.txt', '-cmd',
↪"echo '$MONEY'"]
>>> p = subprocess.Popen(args) # Success!
```

Notez en particulier que les options (comme *-input*) et arguments (comme *eggs.txt*) qui sont séparés par des espaces dans le *shell* iront dans des éléments séparés de la liste, alors que les arguments qui nécessitent des guillemets et échappements quand utilisés dans le *shell* (comme les noms de fichiers contenant des espaces ou la commande *echo* montrée plus haut) forment des éléments uniques.

Sous Windows, si *args* est une séquence, elle sera convertie vers une chaîne de caractères de la manière décrite dans *Convertir une séquence d'arguments vers une chaîne de caractères sous Windows*. Cela fonctionne ainsi parce que la fonction `CreateProcess()` opère sur des chaînes.

Modifié dans la version 3.6 : Le paramètre *args* accepte un *objet chemin* si *shell* vaut `False` et une séquence contenant des objets fichier sur POSIX.

Modifié dans la version 3.8 : Le paramètre *args* accepte un *objet chemin* si *shell* vaut `False` et une séquence contenant des chaines d'octets et des objets chemins sur Windows.

L'argument *shell* (qui vaut `False` par défaut) spécifie s'il faut utiliser un *shell* comme programme à exécuter. Si *shell* vaut `True`, il est recommandé de passer *args* comme une chaîne de caractères plutôt qu'une séquence.

Sur les systèmes POSIX avec `shell=True`, le *shell* par défaut est `/bin/sh`. Si *args* est une chaîne de caractères, la chaîne spécifie la commande à exécuter à travers le *shell*. Cela signifie que la chaîne doit être formatée exactement comme elle le serait si elle était tapée dans l'invite de commandes du *shell*. Cela inclut, par exemple, les guillemets ou les *backslashs* échappant les noms de fichiers contenant des espaces. Si *args* est une séquence, le premier élément spécifie la commande, et les éléments supplémentaires seront traités comme des arguments additionnels à passer au *shell* lui-même. Pour ainsi dire, `Popen` réalise l'équivalent de :

```
Popen(['/bin/sh', '-c', args[0], args[1], ...])
```

Sous Windows avec `shell=True`, la variable d'environnement `COMSPEC` spécifie le *shell* par défaut. La seule raison pour laquelle vous devriez spécifier `shell=True` sous Windows est quand la commande que vous souhaitez exécuter est une *built-in* du *shell* (p. ex. **`dir`** ou **`copy`**). Vous n'avez pas besoin de `shell=True` pour lancer un fichier batch ou un exécutable console.

> **Note**
>
> Lire la section *Security Considerations* avant d'utiliser `shell=True`.

bufsize sera fourni comme l'argument correspondant à la fonction `open()`, lors de la création des objets de fichiers pour les tubes *stdin*/*stdout*/*stderr* :

- `0` means unbuffered (read and write are one system call and can return short)
- `1` means line buffered (only usable if `text=True` or `universal_newlines=True`)
- toutes les autres valeurs positives indiquent d'utiliser un tampon d'approximativement cette taille ;
- un *bufsize* négatif (par défaut) indique au système d'utiliser la valeur par défaut *io.DEFAULT_BUFFER_SIZE*.

Modifié dans la version 3.3.1 : *bufsize* now defaults to -1 to enable buffering by default to match the behavior that most code expects. In versions prior to Python 3.2.4 and 3.3.1 it incorrectly defaulted to `0` which was unbuffered and allowed short reads. This was unintentional and did not match the behavior of Python 2 as most code expected.

L'argument *executable* spécifie un programme de remplacement à exécuter. Il est très rarement nécessaire. Quand `shell=False`, *executable* remplace le programme à exécuter spécifié par *args*. Cependant, les arguments originels d'*args* sont toujours passés au programme. La plupart des programmes traitent le programme spécifié par *args* comme le nom de la commande, qui peut être différent du programme réellement exécuté. Sur les systèmes POSIX, le nom tiré d'*args* devient le nom affiché pour l'exécutable dans des utilitaires tels que **ps**. Si `shell=True`, sur les systèmes POSIX, l'argument *executable* précise le *shell* à utiliser plutôt que `/bin/sh` par défaut.

Modifié dans la version 3.6 : le paramètre *executable* accepte un *objet chemin* sur les systèmes POSIX.

Modifié dans la version 3.8 : le paramètre *executable* accepte un objet *bytes* ou un *objet chemin* sur Windows.

Modifié dans la version 3.12 : Changed Windows shell search order for `shell=True`. The current directory and `%PATH%` are replaced with `%COMSPEC%` and `%SystemRoot%\System32\cmd.exe`. As a result, dropping a malicious program named `cmd.exe` into a current directory no longer works.

stdin, *stdout* and *stderr* specify the executed program's standard input, standard output and standard error file handles, respectively. Valid values are `None`, `PIPE`, `DEVNULL`, an existing file descriptor (a positive integer), and an existing *file object* with a valid file descriptor. With the default settings of `None`, no redirection will occur. `PIPE` indicates that a new pipe to the child should be created. `DEVNULL` indicates that the special file `os.devnull` will be used. Additionally, *stderr* can be `STDOUT`, which indicates that the stderr data from the applications should be captured into the same file handle as for *stdout*.

Si un objet appelable est passé à *preexec_fn*, cet objet sera appelé dans le processus enfant juste avant d'exécuter le programme. (POSIX seulement)

> **Avertissement**
>
> The *preexec_fn* parameter is NOT SAFE to use in the presence of threads in your application. The child process could deadlock before exec is called.

> **Note**
>
> If you need to modify the environment for the child use the *env* parameter rather than doing it in a *preexec_fn*. The *start_new_session* and *process_group* parameters should take the place of code using *preexec_fn* to call `os.setsid()` or `os.setpgid()` in the child.

Modifié dans la version 3.8 : Le paramètre *preexec_fn* n'est plus pris en charge dans les sous-interpréteurs. L'utilisation de ce paramètre lève `RuntimeError`. Cette nouvelle restriction peut affecter les applications déployées avec *mod_wsgi*, *uWSGI* et d'autres environnements qui peuvent embarquer Python.

If *close_fds* is true, all file descriptors except `0`, `1` and `2` will be closed before the child process is execu-

ted. Otherwise when *close_fds* is false, file descriptors obey their inheritable flag as described in *Héritage de descripteurs de fichiers*.

Sur Windows, si *close_fds* est vrai, alors aucun descripteur n'est hérité par le processus enfant à moins d'être explicitement passé dans l'élément `handle_list` de `STARTUPINFO.lpAttributeList`, ou par redirection des descripteurs standards.

Modifié dans la version 3.2 : La valeur par défaut de *close_fds* n'est plus `False`, comme décrit ci-dessus.

Modifié dans la version 3.7 : Sur Windows, la valeur par défaut de *close_fds* a été changée de `False` à `True` lors d'une redirection des descripteurs standards. Il est maintenant possible de donner la valeur `True` à *close_fds* lors d'une redirection de descripteurs standards.

pass_fds est une séquence optionnelle de descripteurs de fichiers à garder ouverts entre le parent et l'enfant. Fournir *pass_fds* force *close_fds* à valoir `True`. (POSIX seulement)

Modifié dans la version 3.2 : Ajout du paramètre *pass_fds*.

If *cwd* is not `None`, the function changes the working directory to *cwd* before executing the child. *cwd* can be a string, bytes or *path-like* object. On POSIX, the function looks for *executable* (or for the first item in *args*) relative to *cwd* if the executable path is a relative path.

Modifié dans la version 3.6 : le paramètre *cwd* accepte un *objet chemin* sur les systèmes POSIX.

Modifié dans la version 3.7 : le paramètre *cwd* accepte un *objet chemin* sur Windows.

Modifié dans la version 3.8 : le paramètre *cwd* accepte une séquence d'octets sur Windows

Si *restore_signals* est vrai (par défaut), tous les signaux que Python a mis à *SIG_IGN* sont restaurés à *SIG_DFL* dans le processus fils avant l'appel à *exec*. Actuellement, cela inclut les signaux *SIGPIPE*, *SIGXFZ* et *SIGXFSZ*. (POSIX seulement)

Modifié dans la version 3.2 : Ajout de *restore_signals*.

If *start_new_session* is true the `setsid()` system call will be made in the child process prior to the execution of the subprocess.

Availability : POSIX

Modifié dans la version 3.2 : Ajout de *start_new_session*.

If *process_group* is a non-negative integer, the `setpgid(0, value)` system call will be made in the child process prior to the execution of the subprocess.

Availability : POSIX

Modifié dans la version 3.11 : *process_group* was added.

If *group* is not `None`, the setregid() system call will be made in the child process prior to the execution of the subprocess. If the provided value is a string, it will be looked up via `grp.getgrnam()` and the value in `gr_gid` will be used. If the value is an integer, it will be passed verbatim. (POSIX only)

Availability : POSIX

Ajouté dans la version 3.9.

If *extra_groups* is not `None`, the setgroups() system call will be made in the child process prior to the execution of the subprocess. Strings provided in *extra_groups* will be looked up via `grp.getgrnam()` and the values in `gr_gid` will be used. Integer values will be passed verbatim. (POSIX only)

Availability : POSIX

Ajouté dans la version 3.9.

If *user* is not `None`, the setreuid() system call will be made in the child process prior to the execution of the subprocess. If the provided value is a string, it will be looked up via `pwd.getpwnam()` and the value in `pw_uid` will be used. If the value is an integer, it will be passed verbatim. (POSIX only)

Availability : POSIX

Ajouté dans la version 3.9.

If *umask* is not negative, the umask() system call will be made in the child process prior to the execution of the subprocess.

Availability : POSIX

Ajouté dans la version 3.9.

If *env* is not `None`, it must be a mapping that defines the environment variables for the new process ; these are used instead of the default behavior of inheriting the current process' environment. This mapping can be str to str on any platform or bytes to bytes on POSIX platforms much like `os.environ` or `os.environb`.

> **Note**
>
> Si spécifié, *env* doit fournir chaque variable requise pour l'exécution du programme. Sous Windows, afin d'exécuter un side-by-side assembly, l'environnement *env* spécifié **doit** contenir une variable `SystemRoot` valide.

If *encoding* or *errors* are specified, or *text* is true, the file objects *stdin*, *stdout* and *stderr* are opened in text mode with the specified *encoding* and *errors*, as described above in *Arguments fréquemment utilisés*. The *universal_newlines* argument is equivalent to *text* and is provided for backwards compatibility. By default, file objects are opened in binary mode.

Ajouté dans la version 3.6 : Ajout d'*encoding* et *errors*.

Ajouté dans la version 3.7 : *text* a été ajouté comme un alias plus lisible de *universal_newlines*.

If given, *startupinfo* will be a `STARTUPINFO` object, which is passed to the underlying `CreateProcess` function.

If given, *creationflags*, can be one or more of the following flags :
— `CREATE_NEW_CONSOLE`
— `CREATE_NEW_PROCESS_GROUP`
— `ABOVE_NORMAL_PRIORITY_CLASS`
— `BELOW_NORMAL_PRIORITY_CLASS`
— `HIGH_PRIORITY_CLASS`
— `IDLE_PRIORITY_CLASS`
— `NORMAL_PRIORITY_CLASS`
— `REALTIME_PRIORITY_CLASS`
— `CREATE_NO_WINDOW`
— `DETACHED_PROCESS`
— `CREATE_DEFAULT_ERROR_MODE`
— `CREATE_BREAKAWAY_FROM_JOB`
pipesize can be used to change the size of the pipe when `PIPE` is used for *stdin*, *stdout* or *stderr*. The size of the pipe is only changed on platforms that support this (only Linux at this time of writing). Other platforms will ignore this parameter.

Modifié dans la version 3.10 : Added the *pipesize* parameter.

Les objets *Popen* sont gérés comme gestionnaires de contexte avec l'instruction `with` : à la sortie, les descripteurs de fichiers standards sont fermés, et le processus est attendu

```
with Popen(["ifconfig"], stdout=PIPE) as proc:
    log.write(proc.stdout.read())
```

Popen et les autres fonctions dans ce module qui l'utilisent lèvent un *évènement d'audit* `subprocess.Popen` avec comme arguments `executable`, `args`, `cwd` et `env`. La valeur pour `args` peut être une simple chaîne de caractères ou une liste de chaînes, en fonction de la plateforme.

Modifié dans la version 3.2 : Ajout de la gestion des gestionnaires de contexte.

Modifié dans la version 3.6 : Le destructeur de *Popen* émet maintenant un avertissement `ResourceWarning` si le processus fils est toujours en cours d'exécution.

Modifié dans la version 3.8 : *Popen* peut utiliser `os.posix_spawn()` dans certains cas pour de meilleures performances. Sur le sous-système Linux de Windows et sur *QEMU* en mode utilisateur, le constructeur de *Popen* utilisant `os.posix_spawn()` ne lève plus d'exception sur les erreurs comme programme manquant, mais le processus enfant échoue avec un `returncode` différent de zéro.

Exceptions

Les exceptions levées dans le processus fils, avant que le nouveau programme ait commencé à s'exécuter, seront ré-levées dans le parent.

The most common exception raised is `OSError`. This occurs, for example, when trying to execute a non-existent file. Applications should prepare for `OSError` exceptions. Note that, when `shell=True`, `OSError` will be raised by the child only if the selected shell itself was not found. To determine if the shell failed to find the requested application, it is necessary to check the return code or output from the subprocess.

Une `ValueError` sera levée si `Popen` est appelé avec des arguments invalides.

`check_call()` et `check_output()` lèveront une `CalledProcessError` si le processus appelé renvoie un code de retour non nul.

All of the functions and methods that accept a *timeout* parameter, such as `run()` and `Popen.communicate()` will raise `TimeoutExpired` if the timeout expires before the process exits.

Toutes les exceptions définies dans ce module héritent de `SubprocessError`.

Ajouté dans la version 3.3 : Ajout de la classe de base `SubprocessError`.

18.6.2 Considérations de sécurité

Unlike some other popen functions, this library will not implicitly choose to call a system shell. This means that all characters, including shell metacharacters, can safely be passed to child processes. If the shell is invoked explicitly, via `shell=True`, it is the application's responsibility to ensure that all whitespace and metacharacters are quoted appropriately to avoid shell injection vulnerabilities. On *some platforms*, it is possible to use `shlex.quote()` for this escaping.

On Windows, batch files (`*.bat` or `*.cmd`) may be launched by the operating system in a system shell regardless of the arguments passed to this library. This could result in arguments being parsed according to shell rules, but without any escaping added by Python. If you are intentionally launching a batch file with arguments from untrusted sources, consider passing `shell=True` to allow Python to escape special characters. See gh-114539 for additional discussion.

18.6.3 Objets *Popen*

Les instances de la classe `Popen` possèdent les méthodes suivantes :

`Popen.`**`poll`**`()`

Vérifie que le processus enfant s'est terminé. Modifie et renvoie l'attribut `returncode`, sinon, renvoie `None`.

`Popen.`**`wait`**`(`*timeout=None*`)`

Attend qu'un processus enfant se termine. Modifie l'attribut `returncode` et le renvoie.

Si le processus ne se termine pas après le nombre de secondes spécifié par *timeout*, une exception `TimeoutExpired` est levée. Cela ne pose aucun problème d'attraper cette exception et de réessayer d'attendre.

> **Note**
>
> Cela provoquera un blocage (*deadlock*) lors de l'utilisation de `stdout=PIPE` ou `stderr=PIPE` si le processus fils génère tellement de données sur le tube qu'il le bloque, en attente que le système d'exploitation permette au tampon du tube d'accepter plus de données. Utilisez `Popen.communicate()` pour éviter ce problème lors de l'utilisation de tubes.

> **Note**
>
> When the `timeout` parameter is not `None`, then (on POSIX) the function is implemented using a busy loop (non-blocking call and short sleeps). Use the `asyncio` module for an asynchronous wait : see `asyncio.create_subprocess_exec`.

Modifié dans la version 3.3 : Ajout de *timeout*.

`Popen.communicate(input=None, timeout=None)`

Interact with process : Send data to stdin. Read data from stdout and stderr, until end-of-file is reached. Wait for process to terminate and set the `returncode` attribute. The optional *input* argument should be data to be sent to the child process, or `None`, if no data should be sent to the child. If streams were opened in text mode, *input* must be a string. Otherwise, it must be bytes.

`communicate()` renvoie une paire `(stdout_data, stderr_data)`. Les données seront des chaînes de caractères si les flux sont ouverts en mode texte, et des objets *bytes* dans le cas contraire.

Notez que si vous souhaitez envoyer des données sur l'entrée standard du processus, vous devez créer l'objet *Popen* avec `stdin=PIPE`. Similairement, pour obtenir autre chose que `None` dans le *n*-uplet résultant, vous devez aussi préciser `stdout=PIPE` et/ou `stderr=PIPE`.

Si le processus ne se termine pas après *timeout* secondes, une exception `TimeoutExpired` est levée. Attraper cette exception et retenter la communication ne fait perdre aucune donnée de sortie.

Le processus enfant n'est pas tué si le *timeout* expire, donc afin de nettoyer proprement le tout, une application polie devrait tuer le processus fils et terminer la communication :

```
proc = subprocess.Popen(...)
try:
    outs, errs = proc.communicate(timeout=15)
except TimeoutExpired:
    proc.kill()
    outs, errs = proc.communicate()
```

> **Note**
>
> Les données lues sont mises en cache en mémoire, donc n'utilisez pas cette méthode si la taille des données est importante voire illimitée.

Modifié dans la version 3.3 : Ajout de *timeout*.

`Popen.send_signal(signal)`

Envoie le signal *signal* au fils.

Do nothing if the process completed.

> **Note**
>
> On Windows, SIGTERM is an alias for `terminate()`. CTRL_C_EVENT and CTRL_BREAK_EVENT can be sent to processes started with a *creationflags* parameter which includes `CREATE_NEW_PROCESS_GROUP`.

`Popen.terminate()`

Stop the child. On POSIX OSs the method sends `SIGTERM` to the child. On Windows the Win32 API function `TerminateProcess()` is called to stop the child.

`Popen.kill()`

Tue le processus fils. Sur les systèmes POSIX, la fonction envoie un signal *SIGKILL* au fils. Sous Windows,

`kill()` est un alias pour `terminate()`.

The following attributes are also set by the class for you to access. Reassigning them to new values is unsupported :

`Popen.`**`args`**

L'argument *args* tel que passé à `Popen` -- une séquence d'arguments du programme ou une simple chaîne de caractères.

Ajouté dans la version 3.3.

`Popen.`**`stdin`**

If the *stdin* argument was `PIPE`, this attribute is a writeable stream object as returned by `open()`. If the *encoding* or *errors* arguments were specified or the *text* or *universal_newlines* argument was `True`, the stream is a text stream, otherwise it is a byte stream. If the *stdin* argument was not `PIPE`, this attribute is `None`.

`Popen.`**`stdout`**

If the *stdout* argument was `PIPE`, this attribute is a readable stream object as returned by `open()`. Reading from the stream provides output from the child process. If the *encoding* or *errors* arguments were specified or the *text* or *universal_newlines* argument was `True`, the stream is a text stream, otherwise it is a byte stream. If the *stdout* argument was not `PIPE`, this attribute is `None`.

`Popen.`**`stderr`**

If the *stderr* argument was `PIPE`, this attribute is a readable stream object as returned by `open()`. Reading from the stream provides error output from the child process. If the *encoding* or *errors* arguments were specified or the *text* or *universal_newlines* argument was `True`, the stream is a text stream, otherwise it is a byte stream. If the *stderr* argument was not `PIPE`, this attribute is `None`.

Avertissement

Utilisez `communicate()` plutôt que `.stdin.write`, `.stdout.read` ou `.stderr.read` pour empêcher les *deadlocks* dus au remplissage des tampons des tubes de l'OS et bloquant le processus enfant.

`Popen.`**`pid`**

L'identifiant de processus du processus enfant.

Notez que si vous passez l'argument *shell* à `True`, il s'agit alors de l'identifiant du *shell* instancié.

`Popen.`**`returncode`**

The child return code. Initially `None`, `returncode` is set by a call to the `poll()`, `wait()`, or `communicate()` methods if they detect that the process has terminated.

A `None` value indicates that the process hadn't yet terminated at the time of the last method call.

Une valeur négative `-N` indique que le processus enfant a été terminé par un signal `N` (seulement sur les systèmes *POSIX*).

18.6.4 Utilitaires *Popen* pour Windows

La classe `STARTUPINFO` et les constantes suivantes sont seulement disponibles sous Windows.

`class` `subprocess.`**`STARTUPINFO`**`(*, dwFlags=0, hStdInput=None, hStdOutput=None, hStdError=None, wShowWindow=0, lpAttributeList=None)`

Une gestion partielle de la structure STARTUPINFO est utilisée lors de la création d'un objet `Popen`. Les attributs ci-dessous peuvent être passés en tant que paramètres *keyword-only*.

Modifié dans la version 3.7 : Ajout de la gestion des paramètres *keyword-only*.

`dwFlags`

Un champ de bits déterminant si certains attributs `STARTUPINFO` sont utilisés quand le processus crée une fenêtre

```
si = subprocess.STARTUPINFO()
si.dwFlags = subprocess.STARTF_USESTDHANDLES | subprocess.STARTF_
↪USESHOWWINDOW
```

hStdInput

Si *dwFlags* spécifie *STARTF_USESTDHANDLES*, cet attribut est le descripteur d'entrée standard du processus. Si *STARTF_USESTDHANDLES* n'est pas spécifié, l'entrée standard par défaut est le tampon du clavier.

hStdOutput

Si *dwFlags* spécifie *STARTF_USESTDHANDLES*, cet attribut est le descripteur de sortie standard du processus. Autrement, l'attribut est ignoré et la sortie standard par défaut est le tampon de la console.

hStdError

Si *dwFlags* spécifie *STARTF_USESTDHANDLES*, cet attribut est le descripteur de sortie d'erreur du processus. Autrement, l'attribut est ignoré et la sortie d'erreur par défaut est le tampon de la console.

wShowWindow

Si *dwFlags* spécifie *STARTF_USESHOWWINDOW*, cet attribut peut-être n'importe quel attribut valide pour le paramètre `nCmdShow` de la fonction ShowWindow, à l'exception de `SW_SHOWDEFAULT`. Autrement, cet attribut est ignoré.

SW_HIDE est fourni pour cet attribut. Il est utilisé quand *Popen* est appelée avec `shell=True`.

lpAttributeList

Dictionnaire des attributs supplémentaires pour la création d'un processus comme donnés dans `STARTUPINFOEX`, lisez UpdateProcThreadAttribute (ressource en anglais).

Attributs gérés :

handle_list

Séquence des descripteurs qui hérités du parent. *close_fds* doit être vrai si la séquence n'est pas vide.

Les descripteurs doivent être temporairement héritables par *os.set_handle_inheritable()* quand ils sont passés au constructeur *Popen*, sinon *OSError* est levée avec l'erreur Windows `ERROR_INVALID_PARAMETER` (87).

> **Avertissement**
>
> Dans un processus avec plusieurs fils d'exécution, faites très attention à éviter la fuite de descripteurs qui sont marqués comme héritables quand vous combinez ceci avec des appels concurrents vers des fonctions de création d'autres processus qui héritent de tous les descripteurs (telle que *os.system()*).

Ajouté dans la version 3.7.

Constantes Windows

Le module *subprocess* expose les constantes suivantes.

`subprocess.`**STD_INPUT_HANDLE**

Le périphérique d'entrée standard. Initialement, il s'agit du tampon de la console d'entrée, `CONIN$`.

`subprocess.`**STD_OUTPUT_HANDLE**

Le périphérique de sortie standard. Initialement, il s'agit du tampon de l'écran de console actif, `CONOUT$`.

`subprocess.`**STD_ERROR_HANDLE**

Le périphérique de sortie d'erreur. Initialement, il s'agit du tampon de l'écran de console actif, `CONOUT$`.

`subprocess.`**SW_HIDE**

Cache la fenêtre. Une autre fenêtre sera activée.

subprocess.**STARTF_USESTDHANDLES**

Spécifie que les attributs *STARTUPINFO.hStdInput*, *STARTUPINFO.hStdOutput* et *STARTUPINFO.hStdError* contiennent des informations additionnelles.

subprocess.**STARTF_USESHOWWINDOW**

Spécifie que l'attribut *STARTUPINFO.wShowWindow* contient des informations additionnelles.

subprocess.**STARTF_FORCEONFEEDBACK**

A *STARTUPINFO.dwFlags* parameter to specify that the *Working in Background* mouse cursor will be displayed while a process is launching. This is the default behavior for GUI processes.

Ajouté dans la version 3.13.

subprocess.**STARTF_FORCEOFFFEEDBACK**

A *STARTUPINFO.dwFlags* parameter to specify that the mouse cursor will not be changed when launching a process.

Ajouté dans la version 3.13.

subprocess.**CREATE_NEW_CONSOLE**

Le nouveau processus instancie une nouvelle console, plutôt que d'hériter de celle de son père (par défaut).

subprocess.**CREATE_NEW_PROCESS_GROUP**

Paramètre `creationflags` de *Popen* pour spécifier si un nouveau groupe de processus doit être créé. Cette option est nécessaire pour utiliser *os.kill()* sur le sous-processus.

L'option est ignorée si *CREATE_NEW_CONSOLE* est spécifié.

subprocess.**ABOVE_NORMAL_PRIORITY_CLASS**

Paramètre `creationflags` de *Popen* pour spécifier qu'un processus aura une priorité au-dessus de la moyenne.

Ajouté dans la version 3.7.

subprocess.**BELOW_NORMAL_PRIORITY_CLASS**

Paramètre `creationflags` de *Popen* pour spécifier qu'un processus aura une priorité au-dessous de la moyenne.

Ajouté dans la version 3.7.

subprocess.**HIGH_PRIORITY_CLASS**

Paramètre `creationflags` de *Popen* pour spécifier qu'un processus aura une priorité haute.

Ajouté dans la version 3.7.

subprocess.**IDLE_PRIORITY_CLASS**

Paramètre `creationflags` de *Popen* pour spécifier qu'un processus aura la priorité la plus basse (inactif ou *idle*).

Ajouté dans la version 3.7.

subprocess.**NORMAL_PRIORITY_CLASS**

A *Popen* `creationflags` parameter to specify that a new process will have a normal priority. (default)

Ajouté dans la version 3.7.

subprocess.**REALTIME_PRIORITY_CLASS**

Paramètre `creationflags` de *Popen* pour spécifier qu'un nouveau processus aura une priorité en temps réel. Vous ne devriez presque JAMAIS utiliser `REALTIME_PRIORITY_CLASS`, car cela interrompt les fils d'exécution système qui gèrent les entrées de la souris, du clavier et *flush* le cache de disque en arrière-plan. Cette classe peut convenir aux applications qui « parlent » directement au matériel ou qui effectuent de brèves tâches nécessitant des interruptions limitées.

Ajouté dans la version 3.7.

subprocess.**CREATE_NO_WINDOW**

Paramètre `creationflags` de `Popen` pour spécifier qu'un processus ne créera pas une nouvelle fenêtre.

Ajouté dans la version 3.7.

subprocess.**DETACHED_PROCESS**

Paramètre `creationflags` de `Popen` pour spécifier qu'un nouveau processus n'héritera pas de la console du processus parent. Cette valeur ne peut pas être utilisée avec CREATE_NEW_CONSOLE.

Ajouté dans la version 3.7.

subprocess.**CREATE_DEFAULT_ERROR_MODE**

Paramètre `creationflags` de `Popen` pour spécifier qu'un nouveau processus n'hérite pas du mode de gestion des erreurs du processus appelant. À la place, le nouveau processus acquiert le mode d'erreur par défaut. Cette fonctionnalité est particulièrement utile pour les applications *shell* avec de multiples fils d'exécution qui s'exécutent avec les erreurs irréversibles désactivées.

Ajouté dans la version 3.7.

subprocess.**CREATE_BREAKAWAY_FROM_JOB**

Paramètre `creationflags` de `Popen` pour spécifier qu'un processus n'est pas associé au *job*.

Ajouté dans la version 3.7.

18.6.5 Ancienne interface (*API*) haut-niveau

Avant Python 3.5, ces trois fonctions représentaient l'API haut-niveau de *subprocess*. Vous pouvez maintenant utiliser `run()` dans de nombreux cas, mais beaucoup de codes existant font appel à ces trois fonctions.

subprocess.**call**(*args*, *, *stdin=None*, *stdout=None*, *stderr=None*, *shell=False*, *cwd=None*, *timeout=None*, ***other_popen_kwargs*)

Lance la commande décrite par *args*, attend qu'elle se termine, et renvoie son attribut `returncode`.

Le code ayant besoin de capturer *stdout* ou *stderr* doit plutôt utiliser :`run()` :

```
run(...).returncode
```

Pour supprimer *stdout* ou *stderr*, passez la valeur `DEVNULL`.

Les arguments montrés plus haut sont sûrement les plus communs. La signature complète de la fonction est en grande partie la même que le constructeur de `Popen` : cette fonction passe tous les arguments fournis autre que *timeout* directement à travers cette interface.

> **Note**
>
> N'utilisez pas `stdout=PIPE` ou `stderr=PIPE` avec cette fonction. Le processus enfant bloquera s'il génère assez de données pour remplir le tampon du tube de l'OS, puisque les tubes ne seront jamais lus.

Modifié dans la version 3.3 : Ajout de *timeout*.

Modifié dans la version 3.12 : Changed Windows shell search order for `shell=True`. The current directory and `%PATH%` are replaced with `%COMSPEC%` and `%SystemRoot%\System32\cmd.exe`. As a result, dropping a malicious program named `cmd.exe` into a current directory no longer works.

subprocess.**check_call**(*args*, *, *stdin=None*, *stdout=None*, *stderr=None*, *shell=False*, *cwd=None*, *timeout=None*, ***other_popen_kwargs*)

Run command with arguments. Wait for command to complete. If the return code was zero then return, otherwise raise `CalledProcessError`. The `CalledProcessError` object will have the return code in the `returncode` attribute. If `check_call()` was unable to start the process it will propagate the exception that was raised.

Le code ayant besoin de capturer *stdout* ou *stderr* doit plutôt utiliser :`run()` :

```
run(..., check=True)
```

Pour supprimer *stdout* ou *stderr*, passez la valeur `DEVNULL`.

Les arguments montrés plus haut sont sûrement les plus communs. La signature complète de la fonction est en grande partie la même que le constructeur de `Popen` : cette fonction passe tous les arguments fournis autre que *timeout* directement à travers cette interface.

> **Note**
>
> N'utilisez pas `stdout=PIPE` ou `stderr=PIPE` avec cette fonction. Le processus enfant bloquera s'il génère assez de données pour remplir le tampon du tube de l'OS, puisque les tubes ne seront jamais lus.

Modifié dans la version 3.3 : Ajout de *timeout*.

Modifié dans la version 3.12 : Changed Windows shell search order for `shell=True`. The current directory and `%PATH%` are replaced with `%COMSPEC%` and `%SystemRoot%\System32\cmd.exe`. As a result, dropping a malicious program named `cmd.exe` into a current directory no longer works.

`subprocess.`**`check_output`**(*args*, **, stdin=None, stderr=None, shell=False, cwd=None, encoding=None, errors=None, universal_newlines=None, timeout=None, text=None, **other_popen_kwargs*)

Lance la commande avec les arguments et renvoie sa sortie.

Si le code de retour est non-nul, la fonction lève une `CalledProcessError`. L'objet `CalledProcessError` contiendra le code de retour dans son attribut `returncode`, et la sortie du programme dans son attribut `output`.

C'est équivalent à :

```
run(..., check=True, stdout=PIPE).stdout
```

The arguments shown above are merely some common ones. The full function signature is largely the same as that of `run()` - most arguments are passed directly through to that interface. One API deviation from `run()` behavior exists : passing `input=None` will behave the same as `input=b''` (or `input=''`, depending on other arguments) rather than using the parent's standard input file handle.

Par défaut, cette fonction renvoie les données encodées sous forme de *bytes*. Le réel encodage des données de sortie peut dépendre de la commande invoquée, donc le décodage du texte devra souvent être géré au niveau de l'application.

Ce comportement peut être redéfini en mettant *text*, *encoding*, *errors*, ou *universal_newlines* à `True` comme décrit dans *Arguments fréquemment utilisés* et `run()`.

Pour capturer aussi la sortie d'erreur dans le résultat, utilisez `stderr=subprocess.STDOUT` :

```
>>> subprocess.check_output(
...     "ls non_existent_file; exit 0",
...     stderr=subprocess.STDOUT,
...     shell=True)
'ls: non_existent_file: No such file or directory\n'
```

Ajouté dans la version 3.1.

Modifié dans la version 3.3 : Ajout de *timeout*.

Modifié dans la version 3.4 : Ajout de la gestion de l'argument nommé *input*.

Modifié dans la version 3.6 : Ajout d'*encoding* et *errors*. Consultez `run()` pour plus d'informations.

Ajouté dans la version 3.7 : *text* a été ajouté comme un alias plus lisible de *universal_newlines*.

Modifié dans la version 3.12 : Changed Windows shell search order for `shell=True`. The current directory

and `%PATH%` are replaced with `%COMSPEC%` and `%SystemRoot%\System32\cmd.exe`. As a result, dropping a malicious program named `cmd.exe` into a current directory no longer works.

18.6.6 Remplacer les fonctions plus anciennes par le module `subprocess`

Dans cette section, « a devient b » signifie que b peut être utilisée en remplacement de a.

> **Note**
>
> Toutes les fonctions « a » dans cette section échouent (plus ou moins) silencieusement si le programme à exécuter ne peut être trouvé ; les fonctions « b » de remplacement lèvent à la place une *OSError*.
>
> De plus, les remplacements utilisant *check_output()* échoueront avec une *CalledProcessError* si l'opération requise produit un code de retour non-nul. La sortie est toujours disponible par l'attribut *output* de l'exception levée.

Dans les exemples suivants, nous supposons que les fonctions utilisées ont déjà été importées depuis le module *subprocess*.

Remplacement de la substitution de commandes de terminal `/bin/sh`

```
output=$(mycmd myarg)
```

devient :

```
output = check_output(["mycmd", "myarg"])
```

Remplacer les *pipes* du *shell*

```
output=$(dmesg | grep hda)
```

devient :

```
p1 = Popen(["dmesg"], stdout=PIPE)
p2 = Popen(["grep", "hda"], stdin=p1.stdout, stdout=PIPE)
p1.stdout.close()  # Allow p1 to receive a SIGPIPE if p2 exits.
output = p2.communicate()[0]
```

L'appel à `p1.stdout.close()` après le démarrage de *p2* est important pour que *p1* reçoive un *SIGPIPE* si *p2* se termine avant lui.

Alternativement, pour des entrées fiables, la gestion des tubes du *shell* peut directement être utilisé :

```
output=$(dmesg | grep hda)
```

devient :

```
output = check_output("dmesg | grep hda", shell=True)
```

Remplacer `os.system()`

```
sts = os.system("mycmd" + " myarg")
# becomes
retcode = call("mycmd" + " myarg", shell=True)
```

Notes :

— Appeler le programme à travers un *shell* n'est habituellement pas requis.
— The *call()* return value is encoded differently to that of *os.system()*.

— The `os.system()` function ignores SIGINT and SIGQUIT signals while the command is running, but the caller must do this separately when using the `subprocess` module.

Un exemple plus réaliste ressemblerait à cela :

```
try:
    retcode = call("mycmd" + " myarg", shell=True)
    if retcode < 0:
        print("Child was terminated by signal", -retcode, file=sys.stderr)
    else:
        print("Child returned", retcode, file=sys.stderr)
except OSError as e:
    print("Execution failed:", e, file=sys.stderr)
```

Remplacer les fonctions de la famille `os.spawn`

Exemple avec *P_NOWAIT* :

```
pid = os.spawnlp(os.P_NOWAIT, "/bin/mycmd", "mycmd", "myarg")
==>
pid = Popen(["/bin/mycmd", "myarg"]).pid
```

Exemple avec *P_WAIT* :

```
retcode = os.spawnlp(os.P_WAIT, "/bin/mycmd", "mycmd", "myarg")
==>
retcode = call(["/bin/mycmd", "myarg"])
```

Exemple avec un tableau :

```
os.spawnvp(os.P_NOWAIT, path, args)
==>
Popen([path] + args[1:])
```

Exemple en passant un environnement :

```
os.spawnlpe(os.P_NOWAIT, "/bin/mycmd", "mycmd", "myarg", env)
==>
Popen(["/bin/mycmd", "myarg"], env={"PATH": "/usr/bin"})
```

Replacing `os.popen()`

La gestion du code de retour se traduit comme suit :

```
pipe = os.popen(cmd, 'w')
...
rc = pipe.close()
if rc is not None and rc >> 8:
    print("There were some errors")
==>
process = Popen(cmd, stdin=PIPE)
...
process.stdin.close()
if process.wait() != 0:
    print("There were some errors")
```

18.6.7 Remplacement des fonctions originales d'invocation du *shell*

Ce module fournit aussi les fonctions suivantes héritées du module `commands` de Python 2.x. Ces opérations invoquent implicitement le *shell* du système et n'apportent aucune des garanties décrites ci-dessus par rapport à la sécurité ou la cohérence de la gestion des exceptions.

`subprocess.`**`getstatusoutput`**(*cmd*, *, *encoding=None*, *errors=None*)

Renvoie les valeurs `(exitcode, output)` de l'exécution de *cmd* dans un *shell*.

Execute the string *cmd* in a shell with `Popen.check_output()` and return a 2-tuple `(exitcode, output)`. *encoding* and *errors* are used to decode output ; see the notes on *Arguments fréquemment utilisés* for more details.

Si la sortie se termine par un caractère de fin de ligne, ce dernier est supprimé. Le code de statut de la commande peut être interprété comme le code de retour de *subprocess*. Par exemple :

```
>>> subprocess.getstatusoutput('ls /bin/ls')
(0, '/bin/ls')
>>> subprocess.getstatusoutput('cat /bin/junk')
(1, 'cat: /bin/junk: No such file or directory')
>>> subprocess.getstatusoutput('/bin/junk')
(127, 'sh: /bin/junk: not found')
>>> subprocess.getstatusoutput('/bin/kill $$')
(-15, '')
```

Availability : Unix, Windows.

Modifié dans la version 3.3.4 : Ajout de la gestion de Windows.

La fonction renvoie maintenant `(exitcode, output)` plutôt que `(status, output)` comme dans les versions de Python 3.3.3 ou antérieures. *exitcode* vaut la même valeur que `returncode`.

Modifié dans la version 3.11 : Added the *encoding* and *errors* parameters.

`subprocess.`**`getoutput`**(*cmd*, *, *encoding=None*, *errors=None*)

Renvoie la sortie (standard et d'erreur) de l'exécution de *cmd* dans un *shell*.

Comme `getstatusoutput()`, à l'exception que le code de statut est ignoré et que la valeur de retour est une chaîne contenant la sortie de la commande. Exemple :

```
>>> subprocess.getoutput('ls /bin/ls')
'/bin/ls'
```

Availability : Unix, Windows.

Modifié dans la version 3.3.4 : Ajout de la gestion de Windows

Modifié dans la version 3.11 : Added the *encoding* and *errors* parameters.

18.6.8 Notes

Convertir une séquence d'arguments vers une chaîne de caractères sous Windows

Sous Windows, une séquence *args* est convertie vers une chaîne qui peut être analysée avec les règles suivantes (qui correspondent aux règles utilisées par l'environnement *MS C*) :

1. Les arguments sont délimités par des espacements, qui peuvent être des espaces ou des tabulations.
2. Une chaîne entourée de double guillemets est interprétée comme un argument seul, qu'elle contienne ou non des espacements. Une chaîne entre guillemets peut être intégrée dans un argument.
3. Un guillemet double précédé d'un *backslash* est interprété comme un guillemet double littéral.
4. Les *backslashes* sont interprétés littéralement, à moins qu'ils précèdent immédiatement un guillemet double.
5. Si des *backslashes* précèdent directement un guillemet double, toute paire de *backslashes* est interprétée comme un *backslash* littéral. Si le nombre de *backslashes* est impair, le dernier *backslash* échappe le prochain guillemet double comme décrit en règle 3.

Voir aussi

shlex
Module qui fournit des fonctions pour analyser et échapper les lignes de commandes.

Disable use of `posix_spawn()`

On Linux, *subprocess* defaults to using the `vfork()` system call internally when it is safe to do so rather than `fork()`. This greatly improves performance.

```
subprocess._USE_POSIX_SPAWN = False  # See CPython issue gh-NNNNNN.
```

It is safe to set this to false on any Python version. It will have no effect on older or newer versions where unsupported. Do not assume the attribute is available to read. Despite the name, a true value does not indicate the corresponding function will be used, only that it may be.

Please file issues any time you have to use these private knobs with a way to reproduce the issue you were seeing. Link to that issue from a comment in your code.

Ajouté dans la version 3.8 : `_USE_POSIX_SPAWN`

18.7 `sched` --- Event scheduler

Code source : Lib/sched.py

The *sched* module defines a class which implements a general purpose event scheduler :

class `sched.`**`scheduler`**(*timefunc=time.monotonic*, *delayfunc=time.sleep*)

The *scheduler* class defines a generic interface to scheduling events. It needs two functions to actually deal with the "outside world" --- *timefunc* should be callable without arguments, and return a number (the "time", in any units whatsoever). The *delayfunc* function should be callable with one argument, compatible with the output of *timefunc*, and should delay that many time units. *delayfunc* will also be called with the argument `0` after each event is run to allow other threads an opportunity to run in multi-threaded applications.

Modifié dans la version 3.3 : *timefunc* and *delayfunc* parameters are optional.

Modifié dans la version 3.3 : *scheduler* class can be safely used in multi-threaded environments.

Exemple :

```
>>> import sched, time
>>> s = sched.scheduler(time.time, time.sleep)
>>> def print_time(a='default'):
...     print("From print_time", time.time(), a)
...
>>> def print_some_times():
...     print(time.time())
...     s.enter(10, 1, print_time)
...     s.enter(5, 2, print_time, argument=('positional',))
...     # despite having higher priority, 'keyword' runs after 'positional' as␣
↪enter() is relative
...     s.enter(5, 1, print_time, kwargs={'a': 'keyword'})
...     s.enterabs(1_650_000_000, 10, print_time, argument=("first enterabs",))
...     s.enterabs(1_650_000_000, 5, print_time, argument=("second enterabs",))
...     s.run()
...     print(time.time())
...
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> print_some_times()
1652342830.3640375
From print_time 1652342830.3642538 second enterabs
From print_time 1652342830.3643398 first enterabs
From print_time 1652342835.3694863 positional
From print_time 1652342835.3696074 keyword
From print_time 1652342840.369612 default
1652342840.3697174
```

18.7.1 Scheduler Objects

`scheduler` instances have the following methods and attributes :

`scheduler.`**`enterabs`**(*time*, *priority*, *action*, *argument=()*, *kwargs={}*)

Schedule a new event. The *time* argument should be a numeric type compatible with the return value of the *timefunc* function passed to the constructor. Events scheduled for the same *time* will be executed in the order of their *priority*. A lower number represents a higher priority.

Executing the event means executing `action(*argument, **kwargs)`. *argument* is a sequence holding the positional arguments for *action*. *kwargs* is a dictionary holding the keyword arguments for *action*.

Return value is an event which may be used for later cancellation of the event (see `cancel()`).

Modifié dans la version 3.3 : *argument* parameter is optional.

Modifié dans la version 3.3 : *kwargs* parameter was added.

`scheduler.`**`enter`**(*delay*, *priority*, *action*, *argument=()*, *kwargs={}*)

Schedule an event for *delay* more time units. Other than the relative time, the other arguments, the effect and the return value are the same as those for `enterabs()`.

Modifié dans la version 3.3 : *argument* parameter is optional.

Modifié dans la version 3.3 : *kwargs* parameter was added.

`scheduler.`**`cancel`**(*event*)

Remove the event from the queue. If *event* is not an event currently in the queue, this method will raise a `ValueError`.

`scheduler.`**`empty`**()

Return `True` if the event queue is empty.

`scheduler.`**`run`**(*blocking=True*)

Run all scheduled events. This method will wait (using the *delayfunc* function passed to the constructor) for the next event, then execute it and so on until there are no more scheduled events.

If *blocking* is false executes the scheduled events due to expire soonest (if any) and then return the deadline of the next scheduled call in the scheduler (if any).

Either *action* or *delayfunc* can raise an exception. In either case, the scheduler will maintain a consistent state and propagate the exception. If an exception is raised by *action*, the event will not be attempted in future calls to `run()`.

If a sequence of events takes longer to run than the time available before the next event, the scheduler will simply fall behind. No events will be dropped ; the calling code is responsible for canceling events which are no longer pertinent.

Modifié dans la version 3.3 : *blocking* parameter was added.

`scheduler.`**`queue`**

Read-only attribute returning a list of upcoming events in the order they will be run. Each event is shown as a *named tuple* with the following fields : time, priority, action, argument, kwargs.

18.8 queue --- A synchronized queue class

Code source : Lib/queue.py

Le module *queue* implémente des files multi-productrices et multi-consommatrices. C'est particulièrement utile en programmation *multi-thread*, lorsque les informations doivent être échangées sans risques entre plusieurs *threads*. La classe *Queue* de ce module implémente tout le verrouillage nécessaire.

The module implements three types of queue, which differ only in the order in which the entries are retrieved. In a FIFO queue, the first tasks added are the first retrieved. In a LIFO queue, the most recently added entry is the first retrieved (operating like a stack). With a priority queue, the entries are kept sorted (using the *heapq* module) and the lowest valued entry is retrieved first.

En interne, ces trois types de files utilisent des verrous pour bloquer temporairement des fils d'exécution concurrents. Cependant, ils n'ont pas été conçus pour être réentrants au sein d'un fil d'exécution.

Le module implémente aussi une FIFO basique, *SimpleQueue*, dont l'implémentation spécialisée fournit plus de garanties au détriment des fonctionnalités.

Le module *queue* définit les classes et les exceptions suivantes :

class queue.**Queue**(*maxsize=0*)

Constructeur pour une file FIFO. *maxsize* est un entier définissant le nombre maximal d'éléments pouvant être mis dans la file. L'insertion sera bloquée lorsque cette borne supérieure sera atteinte, jusqu'à ce que des éléments de la file soient consommés. Si *maxsize* est inférieur ou égal à 0, la taille de la file sera infinie.

class queue.**LifoQueue**(*maxsize=0*)

Constructeur pour une file LIFO. *maxsize* est un entier définissant le nombre maximal d'éléments pouvant être mis dans la file. L'insertion sera bloquée lorsque cette borne supérieure sera atteinte, jusqu'à ce que des éléments de la file soient consommés. Si *maxsize* est inférieur ou égal à 0, la taille de la file sera infinie.

class queue.**PriorityQueue**(*maxsize=0*)

Constructeur pour une file de priorité. *maxsize* est un entier définissant le nombre maximal d'éléments pouvant être mis dans la file. L'insertion sera bloquée lorsque cette borne supérieure sera atteinte, jusqu'à ce que des éléments soient consommés. Si *maxsize* est inférieur ou égal à 0, la taille de la file sera infinie.

The lowest valued entries are retrieved first (the lowest valued entry is the one that would be returned by `min(entries)`). A typical pattern for entries is a tuple in the form : `(priority_number, data)`.

Si les éléments de *data* ne sont pas comparables, les données peuvent être enveloppées dans une classe qui ignore l'élément de données et ne compare que l'ordre de priorité :

```
from dataclasses import dataclass, field
from typing import Any

@dataclass(order=True)
class PrioritizedItem:
    priority: int
    item: Any=field(compare=False)
```

class queue.**SimpleQueue**

Constructeur d'une file illimitée FIFO. Les simples files d'attente ne possèdent pas de fonctionnalités avancées telles que le suivi des tâches.

Ajouté dans la version 3.7.

exception queue.**Empty**

Exception levée lorsque la méthode non bloquante *get()* (ou *get_nowait()*) est appelée sur l'objet *Queue* vide.

exception queue.**Full**

Exception levée lorsque la méthode non bloquante *put()* (ou *put_nowait()*) est appelée sur un objet *Queue* plein.

exception queue.**ShutDown**

Exception raised when *put()* or *get()* is called on a *Queue* object which has been shut down.

Ajouté dans la version 3.13.

18.8.1 Objets Queue

Les objets *Queue* (*Queue*, *LifoQueue* ou *PriorityQueue*) fournissent les méthodes publiques décrites ci-dessous.

Queue.**qsize**()

Renvoie la taille approximative de la file. Notez que `qsize() > 0` ne garantit pas qu'un `get()` ultérieur ne sera pas bloquant et que `qsize() < maxsize` ne garantit pas non plus qu'un `put()` ne sera pas bloquant.

Queue.**empty**()

Renvoie `True` si la file est vide, `False` sinon. Si `empty()` renvoie `True`, cela ne garantit pas qu'un appel ultérieur à `put()` ne sera pas bloquant. Similairement, si `empty()` renvoie `False`, cela ne garantit pas qu'un appel ultérieur à `get()` ne sera pas bloquant.

Queue.**full**()

Renvoie `True` si la file est pleine, `False` sinon. Si `full()` renvoie `True`, cela ne garantit pas qu'un appel ultérieur à `get()` ne sera pas bloquant. Similairement, si `full()` retourne `False`, cela ne garantit pas qu'un appel ultérieur à `put()` ne sera pas bloquant.

Queue.**put**(*item*, *block=True*, *timeout=None*)

Put *item* into the queue. If optional args *block* is true and *timeout* is `None` (the default), block if necessary until a free slot is available. If *timeout* is a positive number, it blocks at most *timeout* seconds and raises the *Full* exception if no free slot was available within that time. Otherwise (*block* is false), put an item on the queue if a free slot is immediately available, else raise the *Full* exception (*timeout* is ignored in that case).

Raises *ShutDown* if the queue has been shut down.

Queue.**put_nowait**(*item*)

Equivalent to `put(item, block=False)`.

Queue.**get**(*block=True*, *timeout=None*)

Retire et renvoie un élément de la file. Si les arguments optionnels *block* et *timeout* valent respectivement `True` et `None` (les valeurs par défaut), la méthode bloque si nécessaire jusqu'à ce qu'un élément soit disponible. Si *timeout* est un entier positif, elle bloque au plus *timeout* secondes et lève l'exception *Empty* s'il n'y avait pas d'élément disponible pendant cette période de temps. Sinon (*block* vaut `False`), elle renvoie un élément s'il y en a un immédiatement disponible. Si ce n'est pas le cas, elle lève l'exception *Empty* (*timeout* est ignoré dans ce cas).

Prior to 3.0 on POSIX systems, and for all versions on Windows, if *block* is true and *timeout* is `None`, this operation goes into an uninterruptible wait on an underlying lock. This means that no exceptions can occur, and in particular a SIGINT will not trigger a *KeyboardInterrupt*.

Raises *ShutDown* if the queue has been shut down and is empty, or if the queue has been shut down immediately.

Queue.**get_nowait**()

Équivalent à `get(False)`.

Deux méthodes sont proposées afin de savoir si les tâches mises dans la file ont été entièrement traitées par les fils d'exécution consommateurs du démon.

`Queue.task_done()`

Indique qu'une tâche précédemment mise dans la file est terminée. Utilisé par les fils d'exécution consommateurs de la file. Pour chaque appel à `get()` effectué afin de récupérer une tâche, un appel ultérieur à `task_done()` informe la file que le traitement de la tâche est terminé.

Si un `join()` est actuellement bloquant, on reprendra lorsque tous les éléments auront été traités (ce qui signifie qu'un appel à `task_done()` a été effectué pour chaque élément qui a été `put()` dans la file).

`shutdown(immediate=True)` calls `task_done()` for each remaining item in the queue.

Lève une exception `ValueError` si appelée plus de fois qu'il y avait d'éléments dans la file.

`Queue.join()`

Bloque jusqu'à ce que tous les éléments de la file aient été obtenus et traités.

The count of unfinished tasks goes up whenever an item is added to the queue. The count goes down whenever a consumer thread calls `task_done()` to indicate that the item was retrieved and all work on it is complete. When the count of unfinished tasks drops to zero, `join()` unblocks.

Exemple montrant comment attendre que les tâches mises dans la file soient terminées :

```
import threading
import queue

q = queue.Queue()

def worker():
    while True:
        item = q.get()
        print(f'Working on {item}')
        print(f'Finished {item}')
        q.task_done()

# Turn-on the worker thread.
threading.Thread(target=worker, daemon=True).start()

# Send thirty task requests to the worker.
for item in range(30):
    q.put(item)

# Block until all tasks are done.
q.join()
print('All work completed')
```

Terminating queues

`Queue` objects can be made to prevent further interaction by shutting them down.

`Queue.shutdown(immediate=False)`

Shut down the queue, making `get()` and `put()` raise `ShutDown`.

By default, `get()` on a shut down queue will only raise once the queue is empty. Set *immediate* to true to make `get()` raise immediately instead.

All blocked callers of `put()` and `get()` will be unblocked. If *immediate* is true, a task will be marked as done for each remaining item in the queue, which may unblock callers of `join()`.

Ajouté dans la version 3.13.

18.8.2 Objets `SimpleQueue`

Les objets `SimpleQueue` fournissent les méthodes publiques décrites ci-dessous.

`SimpleQueue.qsize()`

Renvoie la taille approximative de la file. Notez que `qsize() > 0` ne garantit pas qu'un `get()` ultérieur ne soit pas bloquant.

`SimpleQueue.empty()`

Return `True` if the queue is empty, `False` otherwise. If empty() returns `False` it doesn't guarantee that a subsequent call to get() will not block.

`SimpleQueue.put(item, block=True, timeout=None)`

Met *item* dans la file. La méthode ne bloque jamais et aboutit toujours (sauf en cas de potentielles erreurs de bas niveau, telles qu'un échec d'allocation de mémoire). Les arguments optionnels *block* et *timeout* sont ignorés et fournis uniquement pour la compatibilité avec `Queue.put()`.

Particularité de l'implémentation CPython : This method has a C implementation which is reentrant. That is, a `put()` or `get()` call can be interrupted by another `put()` call in the same thread without deadlocking or corrupting internal state inside the queue. This makes it appropriate for use in destructors such as `__del__` methods or `weakref` callbacks.

`SimpleQueue.put_nowait(item)`

Equivalent to `put(item, block=False)`, provided for compatibility with `Queue.put_nowait()`.

`SimpleQueue.get(block=True, timeout=None)`

Retire et renvoie un élément de la file. Si les arguments optionnels *block* et *timeout* valent respectivement `True` et `None` (les valeurs par défaut), la méthode bloque si nécessaire jusqu'à ce qu'un élément soit disponible. Si *timeout* est un entier positif, elle bloque au plus *timeout* secondes et lève l'exception `Empty` s'il n'y avait pas d'élément disponible pendant cette période de temps. Sinon (*block* vaut `False`), elle renvoie un élément s'il y en a un immédiatement disponible. Si ce n'est pas le cas, elle lève l'exception `Empty` (*timeout* est ignoré dans ce cas).

`SimpleQueue.get_nowait()`

Équivalent à `get(False)`.

Voir aussi

Classe `multiprocessing.Queue`
Une file à utiliser dans un contexte multi-processus (plutôt que *multi-thread*).

`collections.deque` est une implémentation alternative de file non bornée avec des méthodes `append()` et `popleft()` rapides et atomiques ne nécessitant pas de verrouillage et prenant également en charge l'indexation.

18.9 `contextvars` --- Context Variables

Ce module fournit des API pour gérer, stocker et accéder à l'état local de contexte. La classe `ContextVar` est utilisée pour déclarer et travailler avec les *Variables de contexte*. La fonction `copy_context()` et la classe `Context` doivent être utilisées pour la gestion du contexte actuel dans les cadriciels asynchrones.

Context managers that have state should use Context Variables instead of `threading.local()` to prevent their state from bleeding to other code unexpectedly, when used in concurrent code.

Voir aussi **PEP 567** pour plus de détails.

Ajouté dans la version 3.7.

18.9.1 Variables de contexte

class contextvars.**ContextVar**(*name*[, *, *default*])

Cette classe est utilisée pour déclarer une nouvelle variable de contexte, p. ex :

```
var: ContextVar[int] = ContextVar('var', default=42)
```

Le paramètre requis *name* est utilisé à des fins d'introspection et de débogage.

Le paramètre nommé *default* est renvoyé par `ContextVar.get()` quand aucune valeur n'est trouvée dans le contexte actuel pour la variable.

Important : les variables de contexte doivent être créées au plus haut niveau du module et jamais dans des fermetures (*closures*). Les objets `Context` maintiennent des références fortes aux variables de contexte ce qui empêche que les variables de contexte soient correctement nettoyées par le ramasse-miette.

name

Nom de la variable. Cette propriété est en lecture seule.

Ajouté dans la version 3.7.1.

get([*default*])

Renvoie la valeur de la variable de contexte pour le contexte actuel.

S'il n'y a pas de valeur pour la variable dans le contexte actuel, la méthode :
— renvoie la valeur de l'argument *default* passé à la méthode, s'il a été fourni ;
— ou renvoie la valeur par défaut de la variable de contexte, si elle a été créée avec une valeur par défaut ;
— ou lève une erreur `LookupError`.

set(*value*)

Assigne une nouvelle valeur à la variable de contexte dans le contexte actuel.

L'argument requis *value* est la nouvelle valeur pour la variable de contexte.

Renvoie un objet `Token` qui peut être utilisé pour rétablir la variable à sa valeur précédente par la méthode `ContextVar.reset()`.

reset(*token*)

Réinitialise la variable de contexte à la valeur qu'elle avait avant l'appel de `ContextVar.set()` qui a créé le *token*.

Par exemple :

```
var = ContextVar('var')

token = var.set('new value')
# code that uses 'var'; var.get() returns 'new value'.
var.reset(token)

# After the reset call the var has no value again, so
# var.get() would raise a LookupError.
```

class contextvars.**Token**

Les objets *Token* sont renvoyés par la méthode `ContextVar.set()`. Ils peuvent être passés à la méthode `ContextVar.reset()` pour réaffecter la valeur de la variable à ce qu'elle était avant le *set* correspondant.

The token supports context manager protocol to restore the corresponding context variable value at the exit from `with` block :

```
var = ContextVar('var', default='default value')

with var.set('new value'):
    assert var.get() == 'new value'
```

(suite sur la page suivante)

(suite de la page précédente)

```
assert var.get() == 'default value'
```

Ajouté dans la version 3.14.0a5 (unreleased) : Added support for usage as a context manager.

var

Propriété en lecture seule. Pointe vers l'objet `ContextVar` qui a créé le token.

old_value

A read-only property. Set to the value the variable had before the `ContextVar.set()` method call that created the token. It points to `Token.MISSING` if the variable was not set before the call.

MISSING

Objet marqueur utilisé par `Token.old_value`.

18.9.2 Gestion de contexte manuelle

`contextvars.`**`copy_context`**`()`

Renvoie une copie de l'objet `Context` actuel.

Le fragment de code qui suit obtient une copie du contexte actuel et affiche toutes les variables avec leurs valeurs définies dans ce contexte :

```
ctx: Context = copy_context()
print(list(ctx.items()))
```

The function has an $O(1)$ complexity, i.e. works equally fast for contexts with a few context variables and for contexts that have a lot of them.

class `contextvars.`**`Context`**

Tableau associatif entre `ContextVars` et leurs valeurs.

`Context()` crée un contexte vide ne contenant aucune valeur. Pour obtenir une copie du contexte actuel, utilisez la fonction `copy_context()`.

Each thread has its own effective stack of `Context` objects. The *current context* is the `Context` object at the top of the current thread's stack. All `Context` objects in the stacks are considered to be *entered*.

Entering a context, which can be done by calling its `run()` method, makes the context the current context by pushing it onto the top of the current thread's context stack.

Exiting from the current context, which can be done by returning from the callback passed to the `run()` method, restores the current context to what it was before the context was entered by popping the context off the top of the context stack.

Since each thread has its own context stack, `ContextVar` objects behave in a similar fashion to `threading.local()` when values are assigned in different threads.

Attempting to enter an already entered context, including contexts entered in other threads, raises a `RuntimeError`.

After exiting a context, it can later be re-entered (from any thread).

Any changes to `ContextVar` values via the `ContextVar.set()` method are recorded in the current context. The `ContextVar.get()` method returns the value associated with the current context. Exiting a context effectively reverts any changes made to context variables while the context was entered (if needed, the values can be restored by re-entering the context).

Context implémente l'interface `collections.abc.Mapping`.

run(*callable*, **args*, ***kwargs*)

Enters the Context, executes `callable(*args, **kwargs)`, then exits the Context. Returns *callable*'s return value, or propagates an exception if one occurred.

Example :

```
import contextvars

var = contextvars.ContextVar('var')
var.set('spam')
print(var.get())  # 'spam'

ctx = contextvars.copy_context()

def main():
    # 'var' was set to 'spam' before
    # calling 'copy_context()' and 'ctx.run(main)', so:
    print(var.get())  # 'spam'
    print(ctx[var])  # 'spam'

    var.set('ham')

    # Now, after setting 'var' to 'ham':
    print(var.get())  # 'ham'
    print(ctx[var])  # 'ham'

# Any changes that the 'main' function makes to 'var'
# will be contained in 'ctx'.
ctx.run(main)

# The 'main()' function was run in the 'ctx' context,
# so changes to 'var' are contained in it:
print(ctx[var])  # 'ham'

# However, outside of 'ctx', 'var' is still set to 'spam':
print(var.get())  # 'spam'
```

copy()

Renvoie une copie superficielle de l'objet de contexte.

var in context

Renvoie `True` si le *context* a une valeur pour *var* ; sinon renvoie `False`.

context[var]

Renvoie la valeur de la variable `ContextVar` *var*. Si la variable n'est pas définie dans l'objet de contexte, une `KeyError` est levée.

get(*var*[, *default*])

Renvoie la valeur de *var* si *var* possède une valeur dans l'objet de contexte. Renvoie *default* sinon (ou `None` si *default* n'est pas donné).

iter(context)

Renvoie un itérateur sur les variables stockées dans l'objet de contexte.

len(proxy)

Renvoie le nombre de variables définies dans l'objet de contexte.

keys()

Renvoie une liste de toutes les variables dans l'objet de contexte.

values()

Renvoie une liste de toutes les valeurs des variables dans l'objet de contexte.

items()

Renvoie une liste de paires contenant toutes les variables et leurs valeurs dans l'objet de contexte.

18.9.3 Gestion avec *asyncio*

`asyncio` gère nativement les variables de contexte et elles sont prêtes à être utilisées sans configuration supplémentaire. Par exemple, voici un serveur *echo* simple qui utilise une variable de contexte pour que l'adresse d'un client distant soit disponible dans le *Task* qui gère ce client :

```
import asyncio
import contextvars

client_addr_var = contextvars.ContextVar('client_addr')

def render_goodbye():
    # The address of the currently handled client can be accessed
    # without passing it explicitly to this function.

    client_addr = client_addr_var.get()
    return f'Good bye, client @ {client_addr}\r\n'.encode()

async def handle_request(reader, writer):
    addr = writer.transport.get_extra_info('socket').getpeername()
    client_addr_var.set(addr)

    # In any code that we call is now possible to get
    # client's address by calling 'client_addr_var.get()'.

    while True:
        line = await reader.readline()
        print(line)
        if not line.strip():
            break

    writer.write(b'HTTP/1.1 200 OK\r\n')  # status line
    writer.write(b'\r\n')  # headers
    writer.write(render_goodbye())  # body
    writer.close()

async def main():
    srv = await asyncio.start_server(
        handle_request, '127.0.0.1', 8081)

    async with srv:
        await srv.serve_forever()

asyncio.run(main())

# To test it you can use telnet or curl:
#     telnet 127.0.0.1 8081
#     curl 127.0.0.1:8081
```

Les modules suivants servent de fondation pour certains services cités ci-dessus :

18.10 `_thread` --- Low-level threading API

Ce module fournit les primitives de bas niveau pour travailler avec de multiples fils d'exécution (aussi appelés *lightweight processes* ou *tasks*) — plusieurs fils d'exécution de contrôle partagent leur espace de données global. Pour la synchronisation, de simples verrous (aussi appelés des *mutexes* ou des *binary semaphores*) sont fournis. Le module `threading` fournit une API de fils d'exécution de haut niveau, plus facile à utiliser et construite à partir de ce module.

Modifié dans la version 3.7 : Ce module était optionnel, il est maintenant toujours disponible.

Ce module définit les constantes et les fonctions suivantes :

exception _thread.**error**

Levée lors d'erreurs spécifiques aux fils d'exécution.

Modifié dans la version 3.3 : Ceci est à présent un synonyme de l'exception native `RuntimeError`.

_thread.**LockType**

C'est le type d'objets verrous.

_thread.**start_new_thread**(*function*, *args*[, *kwargs*])

Démarre un nouveau fils d'exécution et renvoie son identifiant. Ce fil d'exécution exécute la fonction *function* avec la liste d'arguments *args* (qui doit être un *n*-uplet). L'argument optionnel *kwargs* spécifie un dictionnaire d'arguments nommés.

Au renvoi de la fonction, le fil d'exécution quitte silencieusement.

Lorsque la fonction se termine avec une exception non gérée, `sys.unraisablehook()` est appelée pour gérer cette dernière. L'attribut *object* de l'argument *hook* est *function*. Par défaut, la trace d'appels est affichée puis le fil d'exécution se termine (mais les autres fils d'exécution continuent de s'exécuter).

Lorsque la fonction lève l'exception `SystemExit`, elle est ignorée silencieusement.

Raises an *auditing event* `_thread.start_new_thread` with arguments `function`, `args`, `kwargs`.

Modifié dans la version 3.8 : `sys.unraisablehook()` est maintenant utilisée pour s'occuper des exceptions non gérées.

_thread.**interrupt_main**(*signum=signal.SIGINT*, /)

Simule l'effet d'un signal arrivant au fil d'exécution principal. Un fil d'exécution peut utiliser cette fonction pour interrompre le fil d'exécution principal, bien qu'une interruption immédiate ne soit pas garantie.

If given, *signum* is the number of the signal to simulate. If *signum* is not given, `signal.SIGINT` is simulated.

If the given signal isn't handled by Python (it was set to `signal.SIG_DFL` or `signal.SIG_IGN`), this function does nothing.

Modifié dans la version 3.10 : Ajout du paramètre *signum* pour modifier le numéro du signal.

> **Note**
>
> Cette fonction n'émet pas vraiment le signal, mais programme un appel du gestionnaire associé (à condition qu'il existe). Pour émettre réellement le signal, utilisez `signal.raise_signal()`.

_thread.**exit**()

Lève une exception `SystemExit`. Quand elle n'est pas interceptée, le fil d'exécution se terminera silencieusement.

_thread.**allocate_lock**()

Renvoie un nouveau verrou. Les méthodes des verrous sont décrites ci-dessous. Le verrou est initialement déverrouillé.

_thread.**get_ident**()

Renvoie l'« identifiant de fil » du fil d'exécution courant. C'est un entier non nul. Sa valeur n'a pas de signification directe ; il est destiné à être utilisé comme *cookie* magique, par exemple pour indexer un dictionnaire de données pour chaque fil. Les identifiants de fils peuvent être recyclés lorsqu'un fil d'exécution se termine et qu'un autre fil est créé.

_thread.**get_native_id**()

Renvoie l'identifiant natif complet assigné par le noyau du fil d'exécution actuel. C'est un entier non négatif. Sa valeur peut uniquement être utilisée pour identifier ce fil d'exécution à l'échelle du système (jusqu'à ce que le fil d'exécution se termine, après quoi la valeur peut être recyclée par le système d'exploitation).

Availability : Windows, FreeBSD, Linux, macOS, OpenBSD, NetBSD, AIX, DragonFlyBSD, GNU/kFreeBSD.

Ajouté dans la version 3.8.

Modifié dans la version 3.13 : Added support for GNU/kFreeBSD.

`_thread.stack_size([size])`

Renvoie la taille de la pile d'exécution utilisée lors de la création de nouveaux fils d'exécution. L'argument optionnel *size* spécifie la taille de pile à utiliser pour les fils créés ultérieurement, et doit être à 0 (pour utiliser la taille de la plate-forme ou la valeur configurée par défaut) ou un entier positif supérieur ou égal à 32 768 (32 Kio). Si *size* n'est pas spécifié, 0 est utilisé. Si la modification de la taille de la pile de fils n'est pas prise en charge, une exception `RuntimeError` est levée. Si la taille de la pile spécifiée n'est pas valide, une exception `ValueError` est levée et la taille de la pile n'est pas modifiée. 32 Kio est actuellement la valeur minimale de taille de la pile prise en charge pour garantir un espace de pile suffisant pour l'interpréteur lui-même. Notez que certaines plates-formes peuvent avoir des restrictions particulières sur les valeurs de taille de la pile, telles que l'exigence d'une taille de pile minimale > 32 Kio ou d'une allocation en multiples de la taille de page de la mémoire du système – la documentation de la plate-forme devrait être consultée pour plus d'informations (4 Kio sont courants ; en l'absence de renseignements plus spécifiques, l'approche suggérée est l'utilisation de multiples de 4 096 octets pour la taille de la pile).

Availability : Windows, pthreads.

Unix platforms with POSIX threads support.

`_thread.TIMEOUT_MAX`

The maximum value allowed for the *timeout* parameter of `Lock.acquire`. Specifying a timeout greater than this value will raise an `OverflowError`.

Ajouté dans la version 3.2.

Les verrous ont les méthodes suivantes :

`lock.acquire(blocking=True, timeout=-1)`

Sans aucun argument optionnel, cette méthode acquiert le verrou inconditionnellement, et si nécessaire attend jusqu'à ce qu'il soit relâché par un autre fil d'exécution (un seul fil d'exécution à la fois peut acquérir le verrou — c'est leur raison d'être).

If the *blocking* argument is present, the action depends on its value : if it is false, the lock is only acquired if it can be acquired immediately without waiting, while if it is true, the lock is acquired unconditionally as above.

If the floating-point *timeout* argument is present and positive, it specifies the maximum wait time in seconds before returning. A negative *timeout* argument specifies an unbounded wait. You cannot specify a *timeout* if *blocking* is false.

La valeur renvoyée est `True` si le verrou est acquis avec succès, sinon `False`.

Modifié dans la version 3.2 : Le paramètre *timeout* est nouveau.

Modifié dans la version 3.2 : Le verrou acquis peut maintenant être interrompu par des signaux sur POSIX.

Modifié dans la version 3.14 : Lock acquires can now be interrupted by signals on Windows.

`lock.release()`

Relâche le verrou. Le verrou doit avoir été acquis plus tôt, mais pas nécessairement par le même fil d'exécution.

`lock.locked()`

Renvoie le statut du verrou : `True` s'il a été acquis par certains fils d'exécution, sinon `False`.

En plus de ces méthodes, les objets verrous peuvent aussi être utilisés via l'instruction `with`, e.g :

```
import _thread

a_lock = _thread.allocate_lock()
```

(suite sur la page suivante)

(suite de la page précédente)

```
with a_lock:
    print("a_lock is locked while this executes")
```

Avertissements :

- Interrupts always go to the main thread (the `KeyboardInterrupt` exception will be received by that thread.)
- Appeler la fonction `sys.exit()` ou lever l'exception `SystemExit` est équivalent à appeler la fonction `_thread.exit()`.
- Quand le fil d'exécution principal s'arrête, il est défini par le système si les autres fils d'exécution survivent. Sur beaucoup de systèmes, ils sont tués sans l'exécution des clauses `try… finally` ou l'exécution des destructeurs d'objets.

CHAPITRE 19

Réseau et communication entre processus

Les modules décrits dans ce chapitre fournissent différents mécanismes de mise en réseau et de communication entre processus.

Certains de ces modules ne fonctionnent que pour deux processus sur une seule machine, comme les modules `signal` et `mmap`. D'autres gèrent des protocoles réseaux que deux processus, ou plus, peuvent utiliser pour communiquer entre différentes machines.

La liste des modules documentés dans ce chapitre est :

19.1 `asyncio` --- Asynchronous I/O

Hello World !

```
import asyncio

async def main():
    print('Hello ...')
    await asyncio.sleep(1)
    print('... World!')

asyncio.run(main())
```

`asyncio` est une bibliothèque permettant de faire de la programmation asynchrone en utilisant la syntaxe *async/await*.

`asyncio` constitue la base de nombreux cadriciels (*frameworks*) Python asynchrones qui fournissent des utilitaires réseau et des serveurs web performants, des bibliothèques de connexion à des bases de données, des files d'exécution distribuées, etc.

`asyncio` est souvent le bon choix pour écrire du code réseau de haut-niveau et tributaire des entrées-sorties (*IO-bound*).

`asyncio` fournit des interfaces de programmation **haut-niveau** pour :

- *exécuter des coroutines Python* de manière concurrente et d'avoir le contrôle total sur leur exécution ;
- effectuer *des entrées/sorties réseau et de la communication inter-processus* ;

— contrôler des *sous-processus* ;
— distribuer des tâches avec des *queues* ;
— *synchroniser* du code s'exécutant de manière concurrente ;

En plus, il existe des bibliothèques de **bas-niveau** pour que les *développeurs de bibliothèques et de frameworks* puissent :

— create and manage *event loops*, which provide asynchronous APIs for *networking*, running *subprocesses*, handling *OS signals*, etc ;
— implémenter des protocoles efficaces à l'aide de *transports* ;
— *lier* des bibliothèques basées sur les fonctions de rappel et développer avec la syntaxe *async/await*.

Availability : not WASI.

This module does not work or is not available on WebAssembly. See *Plateformes WebAssembly* for more information.

asyncio REPL

You can experiment with an `asyncio` concurrent context in the *REPL* :

```
$ python -m asyncio
asyncio REPL ...
Use "await" directly instead of "asyncio.run()".
Type "help", "copyright", "credits" or "license" for more information.
>>> import asyncio
>>> await asyncio.sleep(10, result='hello')
'hello'
```

Lève un *évènement d'audit* `cpython.run_stdin` sans argument.

Modifié dans la version 3.12.5 : (also 3.11.10, 3.10.15, 3.9.20, and 3.8.20) Emits audit events.

Modifié dans la version 3.13 : Uses PyREPL if possible, in which case `PYTHONSTARTUP` is also executed. Emits audit events.

Sommaire

19.1.1 Exécuteurs (*runners*)

Code source : Lib/asyncio/runners.py

Cette section décrit les primitives *asyncio* de haut niveau pour exécuter du code asynchrone.

Elles sont construites au-dessus d'une *boucle d'événements* dans le but de simplifier l'utilisation du code asynchrone pour les scénarios les plus courants.

— *Exécution d'un programme asynchrone*
— *Gestionnaire de contexte de l'exécuteur*
— *Gestion de l'interruption par le clavier*

Exécution d'un programme asynchrone

`asyncio.run`(*coro*, *, *debug=None*, *loop_factory=None*)

Execute *coro* in an asyncio event loop and return the result.

The argument can be any awaitable object.

This function runs the awaitable, taking care of managing the asyncio event loop, *finalizing asynchronous generators*, and closing the executor.

Cette fonction ne peut pas être appelée lorsqu'une autre boucle d'événement asynchrone est en cours d'exécution dans le même fil d'exécution.

Si *debug* vaut `True`, la boucle d'événements est exécutée en mode débogage. `False` désactive explicitement le mode débogage. `None` est utilisée pour respecter les paramètres globaux définis par *Mode débogage*.

If *loop_factory* is not `None`, it is used to create a new event loop; otherwise `asyncio.new_event_loop()` is used. The loop is closed at the end. This function should be used as a main entry point for asyncio programs, and should ideally only be called once. It is recommended to use *loop_factory* to configure the event loop instead of policies. Passing `asyncio.EventLoop` allows running asyncio without the policy system.

The executor is given a timeout duration of 5 minutes to shutdown. If the executor hasn't finished within that duration, a warning is emitted and the executor is closed.

Exemple :

```
async def main():
    await asyncio.sleep(1)
    print('hello')

asyncio.run(main())
```

Ajouté dans la version 3.7.

Modifié dans la version 3.9 : mise à jour pour utiliser `loop.shutdown_default_executor()`.

Modifié dans la version 3.10 : *debug* vaut `None` par défaut pour respecter les paramètres du mode de débogage global.

Modifié dans la version 3.12 : Added *loop_factory* parameter.

Modifié dans la version 3.14 : *coro* can be any awaitable object.

> **Note**
>
> The `asyncio` policy system is deprecated and will be removed in Python 3.16; from there on, an explicit *loop_factory* is needed to configure the event loop.

Gestionnaire de contexte de l'exécuteur

class asyncio.**Runner**(*, *debug=None*, *loop_factory=None*)

Gestionnaire de contexte englobant *plusieurs* appels de fonctions asynchrones dans le même contexte.

Parfois, plusieurs fonctions asynchrones de niveau supérieur doivent être appelées dans la même *boucle d'événements* et le même `contextvars.Context`.

Si *debug* vaut `True`, la boucle d'événements est exécutée en mode débogage. `False` désactive explicitement le mode débogage. `None` est utilisée pour respecter les paramètres globaux définis par *Mode débogage*.

loop_factory peut être utilisée pour remplacer la création de la boucle. *loop_factory* a la responsabilité de définir la boucle créée comme boucle courante. Par défaut `asyncio.new_event_loop()` est utilisée et définie comme boucle d'événements actuelle avec `asyncio.set_event_loop()` si *loop_factory* vaut `None`.

Basically, `asyncio.run()` example can be rewritten with the runner usage :

```
async def main():
    await asyncio.sleep(1)
    print('hello')

with asyncio.Runner() as runner:
    runner.run(main())
```

Ajouté dans la version 3.11.

run(*coro*, *, *context=None*)

Execute *coro* in the embedded event loop.

The argument can be any awaitable object.

If the argument is a coroutine, it is wrapped in a Task.

An optional keyword-only *context* argument allows specifying a custom `contextvars.Context` for the code to run in. The runner's default context is used if context is `None`.

Returns the awaitable's result or raises an exception.

Cette fonction ne peut pas être appelée lorsqu'une autre boucle d'événement asynchrone est en cours d'exécution dans le même fil d'exécution.

Modifié dans la version 3.14 : *coro* can be any awaitable object.

close()

Termine l'exécuteur.

Finalise les générateurs asynchrones, arrête l'exécuteur par défaut, ferme la boucle d'événements et libère le `contextvars.Context` en cours.

get_loop()

Renvoie la boucle d'événements associée à l'instance de l'exécuteur.

> **Note**
>
> `Runner` utilise la stratégie d'initialisation paresseuse, son constructeur n'initialise pas les structures de bas niveau sous-jacentes.
>
> La boucle d'événements *loop* et le *context* intégrés sont créés à l'entrée du corps de `with` ou au premier appel de `run()` ou `get_loop()`.

Gestion de l'interruption par le clavier

Ajouté dans la version 3.11.

Lorsque `signal.SIGINT` est déclenché par `Ctrl-C`, l'exception `KeyboardInterrupt` est levée par défaut dans le fils d'exécution principal. Cependant, cela ne fonctionne pas avec `asyncio` car cela peut interrompre le fonctionnement interne de *asyncio* et empêcher le programme de se terminer.

Pour contrer ce problème, `asyncio` gère `signal.SIGINT` comme suit :

1. `asyncio.Runner.run()` installe un gestionnaire `signal.SIGINT` personnalisé avant l'exécution de tout code utilisateur et le supprime à la sortie de la fonction.
2. Le `Runner` crée la tâche principale pour la coroutine transmise pour son exécution.
3. Lorsque `signal.SIGINT` est déclenché par `Ctrl-C`, le gestionnaire de signal personnalisé annule la tâche principale en appelant `asyncio.Task.cancel()` qui lève `asyncio.CancelledError` à l'intérieur de la tâche principale. Cela entraîne la remontée dans la pile Python, les blocs `try/except` et `try/finally` peuvent être utilisés pour le nettoyage des ressources. Une fois la tâche principale annulée, `asyncio.Runner.run()` lève `KeyboardInterrupt`.
4. Un utilisateur peut écrire une boucle tellement petite qu'elle ne peut pas être interrompue par `asyncio.Task.cancel()` ; dans ce cas la seconde suivante `Ctrl-C` lève immédiatement le `KeyboardInterrupt` sans annuler la tâche principale.

19.1.2 Coroutines et tâches

Cette section donne un aperçu des API de haut-niveau du module *asyncio* pour utiliser les coroutines et les tâches.

— *Coroutines*
— Attendables
— *Création de tâches*

— *Annulation de tâche*
— *Groupes de tâches*
— *Attente*
— *Exécution de tâches de manière concurrente*
— *Eager Task Factory*
— *Protection contre l'annulation*
— *Délais d'attente*
— *Primitives d'attente*
— *Exécution dans des fils d'exécution* (threads)
— *Planification depuis d'autres fils d'exécution*
— *Introspection*
— *Objets* Task

Coroutines

Code source : Lib/asyncio/coroutines.py

Les *coroutines* déclarées avec la syntaxe *async/await* sont la manière privilégiée d'écrire des applications asynchrones. Par exemple, l'extrait de code suivant affiche « hello », attend une seconde et affiche ensuite « world » :

```
>>> import asyncio

>>> async def main():
...     print('hello')
...     await asyncio.sleep(1)
...     print('world')

>>> asyncio.run(main())
hello
world
```

Remarquez que simplement appeler une coroutine ne la planifie pas pour exécution :

```
>>> main()
<coroutine object main at 0x1053bb7c8>
```

Pour réellement exécuter une coroutine, *asyncio* fournit les mécanismes suivants :

— La fonction `asyncio.run()` pour exécuter la fonction « main() », le point d'entrée de haut-niveau (voir l'exemple ci-dessus).

— Attendre une coroutine. Le morceau de code suivant attend une seconde, affiche « hello », attend 2 secondes *supplémentaires*, puis affiche enfin « world » :

```
import asyncio
import time

async def say_after(delay, what):
    await asyncio.sleep(delay)
    print(what)

async def main():
    print(f"started at {time.strftime('%X')}")

    await say_after(1, 'hello')
    await say_after(2, 'world')
```

(suite sur la page suivante)

(suite de la page précédente)

```
    print(f"finished at {time.strftime('%X')}")

asyncio.run(main())
```

Sortie attendue :

```
started at 17:13:52
hello
world
finished at 17:13:55
```

— La fonction `asyncio.create_task()` pour exécuter de manière concurrente des coroutines en tant que `tâches` *asyncio*.

Modifions l'exemple ci-dessus et lançons deux coroutines `say_after` *de manière concurrente* :

```
async def main():
    task1 = asyncio.create_task(
        say_after(1, 'hello'))

    task2 = asyncio.create_task(
        say_after(2, 'world'))

    print(f"started at {time.strftime('%X')}")

    # Wait until both tasks are completed (should take
    # around 2 seconds.)
    await task1
    await task2

    print(f"finished at {time.strftime('%X')}")
```

La sortie attendue montre à présent que ce code s'exécute une seconde plus rapidement que le précédent :

```
started at 17:14:32
hello
world
finished at 17:14:34
```

— La classe `asyncio.TaskGroup` fournit une alternative plus moderne à `create_task()`. En utilisant cette API, le dernier exemple devient :

```
async def main():
    async with asyncio.TaskGroup() as tg:
        task1 = tg.create_task(
            say_after(1, 'hello'))

        task2 = tg.create_task(
            say_after(2, 'world'))

        print(f"started at {time.strftime('%X')}")

    # The await is implicit when the context manager exits.

    print(f"finished at {time.strftime('%X')}")
```

Le temps d'exécution et la sortie doivent être les mêmes que pour la version précédente.

Ajouté dans la version 3.11 : `asyncio.TaskGroup`.

Attendables

Un objet est dit *attendable* (*awaitable* en anglais, c.-à-d. qui peut être attendu) s'il peut être utilisé dans une expression `await`. Beaucoup d'API d'*asyncio* sont conçues pour accepter des *attendables*.

Il existe trois types principaux d'*attendables* : les **coroutines**, les **tâches** et les **futurs**.

Coroutines

Les coroutines sont des *awaitables* et peuvent donc être attendues par d'autres coroutines :

```
import asyncio

async def nested():
    return 42

async def main():
    # Nothing happens if we just call "nested()".
    # A coroutine object is created but not awaited,
    # so it *won't run at all*.
    nested()  # will raise a "RuntimeWarning".

    # Let's do it differently now and await it:
    print(await nested())  # will print "42".

asyncio.run(main())
```

Important

dans cette documentation, le terme « coroutine » est utilisé pour désigner deux concepts voisins :
- une *fonction coroutine* : une fonction `async def` ;
- un *objet coroutine* : un objet renvoyé par une *fonction coroutine*.

Tâches

Les *tâches* servent à planifier des coroutines de façon à ce qu'elles s'exécutent de manière concurrente.

Lorsqu'une coroutine est encapsulée dans une *tâche* à l'aide de fonctions comme `asyncio.create_task()`, la coroutine est automatiquement planifiée pour s'exécuter prochainement :

```
import asyncio

async def nested():
    return 42

async def main():
    # Schedule nested() to run soon concurrently
    # with "main()".
    task = asyncio.create_task(nested())

    # "task" can now be used to cancel "nested()", or
    # can simply be awaited to wait until it is complete:
    await task

asyncio.run(main())
```

Futurs

Un `Future` est un objet *attendable* spécial de **bas-niveau**, qui représente le **résultat final** d'une opération asynchrone.

Quand un objet *Future* est *attendu*, cela signifie que la coroutine attendra que ce futur soit résolu à un autre endroit.

Les objets *Future* d'*asyncio* sont nécessaires pour permettre l'exécution de code basé sur les fonctions de rappel avec la syntaxe *async* / *await*.

Il est normalement **inutile** de créer des objets *Future* dans la couche applicative du code.

Les objets *Future*, parfois exposés par des bibliothèques et quelques API d'*asyncio*, peuvent être attendus :

```
async def main():
    await function_that_returns_a_future_object()

    # this is also valid:
    await asyncio.gather(
        function_that_returns_a_future_object(),
        some_python_coroutine()
    )
```

`loop.run_in_executor()` est l'exemple typique d'une fonction bas-niveau renvoyant un objet *Future*.

Création de tâches

Code source : Lib/asyncio/tasks.py

`asyncio.`**`create_task`**(*coro*, *, *name=None*, *context=None*)

Encapsule la *coroutine* *coro* dans une tâche et la planifie pour exécution. Renvoie l'objet `Task`.

Si *name* n'est pas `None`, il est défini comme le nom de la tâche en utilisant `Task.set_name()`.

L'argument (uniquement nommé) facultatif *context* permet de spécifier un `contextvars.Context` personnalisé pour la coroutine à exécuter. La copie de contexte actuelle est créée lorsqu'aucun *context* n'est fourni.

La tâche est exécutée dans la boucle renvoyée par `get_running_loop()` ; `RuntimeError` est levée s'il n'y a pas de boucle en cours d'exécution dans le fil actuel.

> **Note**
>
> `asyncio.TaskGroup.create_task()` is a new alternative leveraging structural concurrency ; it allows for waiting for a group of related tasks with strong safety guarantees.

> **Important**
>
> gardez une référence au résultat de cette fonction, pour éviter qu'une tâche ne disparaisse en cours d'exécution. La boucle d'événements ne conserve que les références faibles aux tâches. Une tâche qui n'est pas référencée ailleurs peut être supprimée par le ramasse-miettes à tout moment, même avant qu'elle ne soit terminée. Pour créer des tâches d'arrière-plan fiables de type « lance et oublie », rassemblez-les dans une collection :
>
> ```
> background_tasks = set()
>
> for i in range(10):
>  task = asyncio.create_task(some_coro(param=i))
>
>  # Add task to the set. This creates a strong reference.
>  background_tasks.add(task)
>
>  # To prevent keeping references to finished tasks forever,
>  # make each task remove its own reference from the set after
>  # completion:
>  task.add_done_callback(background_tasks.discard)
> ```

Ajouté dans la version 3.7.

Modifié dans la version 3.8 : ajout du paramètre *name*.

Modifié dans la version 3.11 : ajout du paramètre *context*.

Annulation de tâche

Les tâches peuvent être annulées facilement et en toute sécurité. Lorsqu'une tâche est annulée, `asyncio.CancelledError` est levée dans la tâche à la première occasion.

Il est recommandé que les coroutines utilisent des blocs `try/finally` pour exécuter de manière robuste la logique de nettoyage. Dans le cas où `asyncio.CancelledError` est explicitement interceptée, elle devrait généralement être propagée lorsque le nettoyage est terminé. `asyncio.CancelledError` sous-classe directement `BaseException` donc la plupart du code n'a pas besoin d'en être conscient.

Les composants *asyncio* qui permettent la concurrence structurée, comme `asyncio.TaskGroup` et `asyncio.timeout()`, sont implémentés en utilisant l'annulation en interne et peuvent mal se comporter si une coroutine ne propage pas `asyncio.CancelledError`. De même, le code utilisateur ne doit généralement pas appeler `uncancel`. Cependant, dans les cas où la suppression de `asyncio.CancelledError` est vraiment souhaitée, il est également nécessaire d'appeler `uncancel()` pour supprimer complètement l'état d'annulation.

Groupes de tâches

Les groupes de tâches combinent une API de création de tâches avec un moyen pratique et fiable d'attendre la fin de toutes les tâches du groupe.

class asyncio.**TaskGroup**

Gestionnaire de contexte asynchrone responsable d'un groupe de tâches. Des tâches peuvent être ajoutées au groupe en utilisant `create_task()`. Toutes les tâches sont attendues à la sortie du gestionnaire de contexte.

Ajouté dans la version 3.11.

create_task(*coro*, ***, *name=None*, *context=None*)

Create a task in this task group. The signature matches that of `asyncio.create_task()`. If the task group is inactive (e.g. not yet entered, already finished, or in the process of shutting down), we will close the given `coro`.

Modifié dans la version 3.13 : Close the given coroutine if the task group is not active.

Exemple :

```
async def main():
    async with asyncio.TaskGroup() as tg:
        task1 = tg.create_task(some_coro(...))
        task2 = tg.create_task(another_coro(...))
    print(f"Both tasks have completed now: {task1.result()}, {task2.result()}")
```

L'instruction `async with` attend la fin de toutes les tâches du groupe. Lors de l'attente, de nouvelles tâches peuvent encore être ajoutées au groupe (par exemple, en passant `tg` dans l'une des coroutines et en appelant `tg.create_task()` dans cette coroutine). Une fois que la dernière tâche est terminée et que le bloc `async with` est quitté, aucune nouvelle tâche ne peut être ajoutée au groupe.

La première fois que l'une des tâches appartenant au groupe échoue avec une exception autre que `asyncio.CancelledError`, les tâches restantes du groupe sont annulées. Aucune autre tâche ne peut alors être ajoutée au groupe. À ce stade, si le corps de l'instruction `async with` est toujours actif (par exemple, `__aexit__()` n'a pas encore été appelé), la tâche contenant directement l'instruction `async with` est également annulée. Le résultat `asyncio.CancelledError` interrompt un `await`, mais il ne sort pas de l'instruction `async with` englobante.

Une fois toutes les tâches terminées, si des tâches ont échoué avec une exception autre que `asyncio.CancelledError`, ces exceptions sont combinées dans un `ExceptionGroup` ou `BaseExceptionGroup` (selon le cas ; voir leur documentation) qui est ensuite levé.

Deux exceptions de base sont traitées spécialement : si une tâche échoue avec *KeyboardInterrupt* ou *SystemExit*, le groupe de tâches annule toujours les tâches restantes et les attend, mais alors la *KeyboardInterrupt* ou la *SystemExit* initiale est levée à nouveau au lieu de *ExceptionGroup* ou *BaseExceptionGroup*.

Si le corps de l'instruction `async with` se termine avec une exception (donc `__aexit__()` est appelé avec un ensemble d'exceptions), cela est traité de la même manière que si l'une des tâches échouait : les tâches restantes sont annulées puis attendues, et les exceptions de non-annulation sont regroupées dans un groupe d'exceptions et levées. L'exception transmise à `__aexit__()`, à moins qu'il ne s'agisse de *asyncio.CancelledError*, est également incluse dans le groupe d'exceptions. Le même cas spécial concerne *KeyboardInterrupt* et *SystemExit* comme dans le paragraphe précédent.

Task groups are careful not to mix up the internal cancellation used to "wake up" their `__aexit__()` with cancellation requests for the task in which they are running made by other parties. In particular, when one task group is syntactically nested in another, and both experience an exception in one of their child tasks simultaneously, the inner task group will process its exceptions, and then the outer task group will receive another cancellation and process its own exceptions.

In the case where a task group is cancelled externally and also must raise an *ExceptionGroup*, it will call the parent task's *cancel()* method. This ensures that a *asyncio.CancelledError* will be raised at the next `await`, so the cancellation is not lost.

Task groups preserve the cancellation count reported by *asyncio.Task.cancelling()*.

Modifié dans la version 3.13 : Improved handling of simultaneous internal and external cancellations and correct preservation of cancellation counts.

Terminating a Task Group

While terminating a task group is not natively supported by the standard library, termination can be achieved by adding an exception-raising task to the task group and ignoring the raised exception :

```
import asyncio
from asyncio import TaskGroup

class TerminateTaskGroup(Exception):
    """Exception raised to terminate a task group."""

async def force_terminate_task_group():
    """Used to force termination of a task group."""
    raise TerminateTaskGroup()

async def job(task_id, sleep_time):
    print(f'Task {task_id}: start')
    await asyncio.sleep(sleep_time)
    print(f'Task {task_id}: done')

async def main():
    try:
        async with TaskGroup() as group:
            # spawn some tasks
            group.create_task(job(1, 0.5))
            group.create_task(job(2, 1.5))
            # sleep for 1 second
            await asyncio.sleep(1)
            # add an exception-raising task to force the group to terminate
            group.create_task(force_terminate_task_group())
    except* TerminateTaskGroup:
        pass
```

(suite sur la page suivante)

(suite de la page précédente)

```
asyncio.run(main())
```

Expected output :

```
Task 1: start
Task 2: start
Task 1: done
```

Attente

coroutine asyncio.**sleep**(*delay*, *result=None*)

Attend pendant *delay* secondes.

Si *result* est spécifié, il est renvoyé à l'appelant quand la coroutine se termine.

`sleep()` suspend systématiquement la tâche courante, ce qui permet aux autres tâches de s'exécuter.

Définir le délai sur 0 fournit un chemin optimisé pour permettre à d'autres tâches de s'exécuter. Cela peut être utilisé par les fonctions de longue durée pour éviter de bloquer la boucle d'événements pendant toute la durée de l'appel de fonction.

Exemple d'une coroutine affichant la date toutes les secondes pendant 5 secondes :

```
import asyncio
import datetime

async def display_date():
    loop = asyncio.get_running_loop()
    end_time = loop.time() + 5.0
    while True:
        print(datetime.datetime.now())
        if (loop.time() + 1.0) >= end_time:
            break
        await asyncio.sleep(1)

asyncio.run(display_date())
```

Modifié dans la version 3.10 : le paramètre *loop* a été enlevé.

Modifié dans la version 3.13 : Raises `ValueError` if *delay* is `nan`.

Exécution de tâches de manière concurrente

awaitable asyncio.**gather**(**aws*, *return_exceptions=False*)

Exécute les objets *awaitable* de la séquence *aws*, *de manière concurrente*.

Si un *attendable* de *aws* est une coroutine, celui-ci est automatiquement planifié comme une tâche *Task*.

Si tous les *awaitables* s'achèvent avec succès, le résultat est la liste des valeurs renvoyées. L'ordre de cette liste correspond à l'ordre des *awaitables* dans *aws*.

Si *return_exceptions* vaut `False` (valeur par défaut), la première exception levée est immédiatement propagée vers la tâche en attente dans le `gather()`. Les autres *attendables* dans la séquence *aws* **ne sont pas annulés** et poursuivent leur exécution.

Si *return_exceptions* vaut `True`, les exceptions sont traitées de la même manière que les exécutions normales, et incluses dans la liste des résultats.

Si `gather()` est *annulé*, tous les *awaitables* en cours (ceux qui n'ont pas encore fini de s'exécuter) sont également *annulés*.

Si n'importe quel *Task* ou *Future* de la séquence *aws* est *annulé*, il est traité comme s'il avait levé `CancelledError` — l'appel à `gather()` n'est alors **pas** annulé. Ceci permet d'empêcher que l'annulation d'une tâche ou d'un futur entraîne l'annulation des autres tâches ou futurs.

> **Note**
>
> A new alternative to create and run tasks concurrently and wait for their completion is `asyncio.TaskGroup`. *TaskGroup* provides stronger safety guarantees than *gather* for scheduling a nesting of subtasks : if a task (or a subtask, a task scheduled by a task) raises an exception, *TaskGroup* will, while *gather* will not, cancel the remaining scheduled tasks).

Exemple :

```
import asyncio

async def factorial(name, number):
    f = 1
    for i in range(2, number + 1):
        print(f"Task {name}: Compute factorial({number}), currently i={i}...")
        await asyncio.sleep(1)
        f *= i
    print(f"Task {name}: factorial({number}) = {f}")
    return f

async def main():
    # Schedule three calls *concurrently*:
    L = await asyncio.gather(
        factorial("A", 2),
        factorial("B", 3),
        factorial("C", 4),
    )
    print(L)

asyncio.run(main())

# Expected output:
#
#     Task A: Compute factorial(2), currently i=2...
#     Task B: Compute factorial(3), currently i=2...
#     Task C: Compute factorial(4), currently i=2...
#     Task A: factorial(2) = 2
#     Task B: Compute factorial(3), currently i=3...
#     Task C: Compute factorial(4), currently i=3...
#     Task B: factorial(3) = 6
#     Task C: Compute factorial(4), currently i=4...
#     Task C: factorial(4) = 24
#     [2, 6, 24]
```

> **Note**
>
> If *return_exceptions* is false, cancelling gather() after it has been marked done won't cancel any submitted awaitables. For instance, gather can be marked done after propagating an exception to the caller, therefore, calling `gather.cancel()` after catching an exception (raised by one of the awaitables) from gather won't cancel any other awaitables.

Modifié dans la version 3.7 : Si *gather* est lui-même annulé, l'annulation est propagée indépendamment de

return_exceptions.

Modifié dans la version 3.10 : le paramètre *loop* a été enlevé.

Obsolète depuis la version 3.10 : Un avertissement d'obsolescence est émis si aucun argument positionnel n'est fourni ou si tous les arguments positionnels ne sont pas des objets de type Future et qu'il n'y a pas de boucle d'événements en cours d'exécution.

Eager Task Factory

`asyncio.`**`eager_task_factory`**(*loop*, *coro*, *, *name=None*, *context=None*)

A task factory for eager task execution.

When using this factory (via `loop.set_task_factory(asyncio.eager_task_factory)`), coroutines begin execution synchronously during `Task` construction. Tasks are only scheduled on the event loop if they block. This can be a performance improvement as the overhead of loop scheduling is avoided for coroutines that complete synchronously.

A common example where this is beneficial is coroutines which employ caching or memoization to avoid actual I/O when possible.

> **Note**
>
> Immediate execution of the coroutine is a semantic change. If the coroutine returns or raises, the task is never scheduled to the event loop. If the coroutine execution blocks, the task is scheduled to the event loop. This change may introduce behavior changes to existing applications. For example, the application's task execution order is likely to change.

Ajouté dans la version 3.12.

`asyncio.`**`create_eager_task_factory`**(*custom_task_constructor*)

Create an eager task factory, similar to `eager_task_factory()`, using the provided *custom_task_constructor* when creating a new task instead of the default `Task`.

custom_task_constructor must be a *callable* with the signature matching the signature of `Task.__init__`. The callable must return a `asyncio.Task`-compatible object.

This function returns a *callable* intended to be used as a task factory of an event loop via `loop.set_task_factory(factory)`).

Ajouté dans la version 3.12.

Protection contre l'annulation

`awaitable` `asyncio.`**`shield`**(*aw*)

Empêche qu'un objet *awaitable* puisse être `annulé`.

Si *aw* est une coroutine, elle est planifiée automatiquement comme une tâche.

L'instruction :

```
task = asyncio.create_task(something())
res = await shield(task)
```

est équivalente à :

```
res = await something()
```

à la différence près que, si la coroutine qui la contient est annulée, la tâche s'exécutant dans `something()` n'est pas annulée. Du point de vue de `something()`, il n'y a pas eu d'annulation. Cependant, son appelant est bien annulé, donc l'expression *await* lève bien une `CancelledError`.

Si `something()` est annulée d'une autre façon (c.-à-d. depuis elle-même) ceci annule également `shield()`.

Pour ignorer complètement l'annulation (déconseillé), la fonction `shield()` peut être combinée à une clause *try* / *except*, comme dans le code ci-dessous :

```
task = asyncio.create_task(something())
try:
    res = await shield(task)
except CancelledError:
    res = None
```

Important

sauvegardez une référence aux tâches passées à cette fonction, pour éviter qu'une tâche ne disparaisse en cours d'exécution. La boucle d'événements ne conserve que les références faibles aux tâches. Une tâche qui n'est pas référencée ailleurs peut faire l'objet d'une suppression par le ramasse-miettes à tout moment, même avant qu'elle ne soit terminée.

Modifié dans la version 3.10 : le paramètre *loop* a été enlevé.

Obsolète depuis la version 3.10 : un avertissement d'obsolescence est émis si *aw* n'est pas un objet de type *future* et qu'il n'y a pas de boucle d'événement en cours d'exécution.

Délais d'attente

`asyncio.timeout(delay)`

Return an asynchronous context manager that can be used to limit the amount of time spent waiting on something.

delay peut-être soit `None`, soit le nombre de secondes (entier ou décimal) d'attente. Si *delay* vaut `None`, aucune limite n'est appliquée ; cela peut être utile lorsque le délai d'attente est inconnu au moment de la création du gestionnaire de contexte.

Dans les deux cas, le gestionnaire de contexte peut être redéfini après sa création en utilisant `Timeout.reschedule()`.

Exemple :

```
async def main():
    async with asyncio.timeout(10):
        await long_running_task()
```

If `long_running_task` takes more than 10 seconds to complete, the context manager will cancel the current task and handle the resulting `asyncio.CancelledError` internally, transforming it into a `TimeoutError` which can be caught and handled.

Note

The `asyncio.timeout()` context manager is what transforms the `asyncio.CancelledError` into a `TimeoutError`, which means the `TimeoutError` can only be caught *outside* of the context manager.

Example of catching `TimeoutError` :

```
async def main():
    try:
        async with asyncio.timeout(10):
            await long_running_task()
    except TimeoutError:
        print("The long operation timed out, but we've handled it.")
```

(suite sur la page suivante)

(suite de la page précédente)

```
    print("This statement will run regardless.")
```

Le gestionnaire de contexte produit par `asyncio.timeout()` peut être reprogrammé à une échéance différente et inspecté.

class asyncio.**Timeout**(*when*)

Gestionnaire de contexte asynchrone pour annuler les coroutines en retard.

`when` doit être un temps absolu au bout duquel le contexte doit expirer, tel que mesuré par l'horloge de la boucle d'événements :

— Si `when` vaut `None`, le délai maximal ne se déclenche jamais.

— Si `when < loop.time()`, le délai maximal se déclenche à la prochaine itération de la boucle d'événement.

when() → *float* | *None*

Renvoie la limite de temps d'exécution définie actuellement, ou `None` s'il n'y en a pas.

reschedule(*when : float* | None)

Reprogramme le délai d'attente.

expired() → *bool*

Renvoie si le gestionnaire de contexte a dépassé son délai (c.-à-d. s'il a expiré).

Exemple :

```
async def main():
    try:
        # We do not know the timeout when starting, so we pass ``None``.
        async with asyncio.timeout(None) as cm:
            # We know the timeout now, so we reschedule it.
            new_deadline = get_running_loop().time() + 10
            cm.reschedule(new_deadline)

            await long_running_task()
    except TimeoutError:
        pass

    if cm.expired():
        print("Looks like we haven't finished on time.")
```

Les gestionnaires de contexte de délai maximal peuvent être imbriqués en toute sécurité.

Ajouté dans la version 3.11.

asyncio.**timeout_at**(*when*)

Semblable à `asyncio.timeout()`, sauf que *when* est le temps absolu pour arrêter d'attendre, ou `None`.

Exemple :

```
async def main():
    loop = get_running_loop()
    deadline = loop.time() + 20
    try:
        async with asyncio.timeout_at(deadline):
            await long_running_task()
    except TimeoutError:
        print("The long operation timed out, but we've handled it.")

    print("This statement will run regardless.")
```

Ajouté dans la version 3.11.

coroutine asyncio.**wait_for**(*aw*, *timeout*)

Attend la fin de l'*awaitable* *aw* avec délai d'attente.

Si *aw* est une coroutine, elle est planifiée automatiquement comme une tâche.

timeout peut-être soit `None`, soit le nombre de secondes (entier ou décimal) d'attente. Si *timeout* vaut `None`, la fonction s'interrompt jusqu'à ce que le futur s'achève.

Si le délai d'attente maximal est dépassé, la tâche est annulée et l'exception `TimeoutError` est levée.

Pour empêcher `l'annulation` de la tâche, il est nécessaire de l'encapsuler dans une fonction `shield()`.

Cette fonction attend que le futur soit réellement annulé, donc le temps d'attente total peut être supérieur à *timeout*. Si une exception se produit lors de l'annulation, elle est propagée.

Si l'attente est annulée, le futur *aw* est également annulé.

Exemple :

```
async def eternity():
    # Sleep for one hour
    await asyncio.sleep(3600)
    print('yay!')

async def main():
    # Wait for at most 1 second
    try:
        await asyncio.wait_for(eternity(), timeout=1.0)
    except TimeoutError:
        print('timeout!')

asyncio.run(main())

# Expected output:
#
#     timeout!
```

Modifié dans la version 3.7 : Si le dépassement du délai d'attente maximal provoque l'annulation de *aw*, `wait_for` attend que *aw* soit annulée. Auparavant, l'exception `TimeoutError` était immédiatement levée.

Modifié dans la version 3.10 : le paramètre *loop* a été enlevé.

Modifié dans la version 3.11 : Raises `TimeoutError` instead of `asyncio.TimeoutError`.

Primitives d'attente

coroutine asyncio.**wait**(*aws*, *, *timeout=None*, *return_when=ALL_COMPLETED*)

Exécute les instances `Future` et `Task` de l'itérable *aws* de manière concurrente, et s'interrompt jusqu'à ce que la condition décrite dans *return_when* soit vraie.

The *aws* iterable must not be empty.

Renvoie deux ensembles de *Tasks* / *Futures* : `(done, pending)`.

Utilisation :

```
done, pending = await asyncio.wait(aws)
```

timeout (entier ou décimal), si précisé, peut-être utilisé pour contrôler le nombre maximal de secondes d'attente avant de se terminer.

Cette fonction ne lève pas `TimeoutError`. Les futurs et les tâches qui ne sont pas finis quand le délai d'attente maximal est dépassé sont tout simplement renvoyés dans le second ensemble.

return_when indique quand la fonction doit se terminer. Il peut prendre les valeurs suivantes :

Constante	Description
asyncio.**FIRST_COMPLETED**	La fonction se termine lorsque n'importe quel futur se termine ou est annulé.
asyncio.**FIRST_EXCEPTION**	The function will return when any future finishes by raising an exception. If no future raises an exception then it is equivalent to *ALL_COMPLETED*.
asyncio.**ALL_COMPLETED**	La fonction se termine lorsque les *futurs* sont tous finis ou annulés.

À la différence de *wait_for()*, `wait()` n'annule pas les futurs quand le délai d'attente est dépassé.

Modifié dans la version 3.10 : le paramètre *loop* a été enlevé.

Modifié dans la version 3.11 : Passer directement des objets coroutines à `wait()` est interdit.

Modifié dans la version 3.12 : Added support for generators yielding tasks.

asyncio.**as_completed**(*aws*, *, *timeout=None*)

Run *awaitable objects* in the *aws* iterable concurrently. The returned object can be iterated to obtain the results of the awaitables as they finish.

The object returned by `as_completed()` can be iterated as an *asynchronous iterator* or a plain *iterator*. When asynchronous iteration is used, the originally-supplied awaitables are yielded if they are tasks or futures. This makes it easy to correlate previously-scheduled tasks with their results. Example :

```
ipv4_connect = create_task(open_connection("127.0.0.1", 80))
ipv6_connect = create_task(open_connection("::1", 80))
tasks = [ipv4_connect, ipv6_connect]

async for earliest_connect in as_completed(tasks):
    # earliest_connect is done. The result can be obtained by
    # awaiting it or calling earliest_connect.result()
    reader, writer = await earliest_connect

    if earliest_connect is ipv6_connect:
        print("IPv6 connection established.")
    else:
        print("IPv4 connection established.")
```

During asynchronous iteration, implicitly-created tasks will be yielded for supplied awaitables that aren't tasks or futures.

When used as a plain iterator, each iteration yields a new coroutine that returns the result or raises the exception of the next completed awaitable. This pattern is compatible with Python versions older than 3.13 :

```
ipv4_connect = create_task(open_connection("127.0.0.1", 80))
ipv6_connect = create_task(open_connection("::1", 80))
tasks = [ipv4_connect, ipv6_connect]

for next_connect in as_completed(tasks):
    # next_connect is not one of the original task objects. It must be
    # awaited to obtain the result value or raise the exception of the
    # awaitable that finishes next.
    reader, writer = await next_connect
```

A *TimeoutError* is raised if the timeout occurs before all awaitables are done. This is raised by the `async`

`for` loop during asynchronous iteration or by the coroutines yielded during plain iteration.

Modifié dans la version 3.10 : le paramètre *loop* a été enlevé.

Obsolète depuis la version 3.10 : Un avertissement d'obsolescence est émis si tous les objets en attente dans l'itérable *aws* ne sont pas des objets de type *Future* et qu'il n'y a pas de boucle d'événement en cours d'exécution.

Modifié dans la version 3.12 : Added support for generators yielding tasks.

Modifié dans la version 3.13 : The result can now be used as either an *asynchronous iterator* or as a plain *iterator* (previously it was only a plain iterator).

Exécution dans des fils d'exécution (*threads*)

coroutine `asyncio.`**`to_thread`**(*func*, /, **args*, ***kwargs*)

Exécute la fonction *func* de manière asynchrone dans un fil d'exécution séparé.

Tous les **args* et ***kwargs* fournis à cette fonction sont directement passés à *func*. De plus, le `contextvars.Context` actuel est propagé, ce qui permet d'accéder aux variables de contexte du fil de boucle d'événements dans le fil séparé.

Renvoie une coroutine qui peut être attendue pour obtenir le résultat éventuel de *func*.

Cette fonction coroutine est principalement destinée à être utilisée pour exécuter des fonctions/méthodes faisant beaucoup d'entrées-sorties et qui bloqueraient autrement la boucle d'événements si elles étaient exécutées dans le fil d'exécution principal. Par exemple :

```
def blocking_io():
    print(f"start blocking_io at {time.strftime('%X')}")
    # Note that time.sleep() can be replaced with any blocking
    # IO-bound operation, such as file operations.
    time.sleep(1)
    print(f"blocking_io complete at {time.strftime('%X')}")

async def main():
    print(f"started main at {time.strftime('%X')}")

    await asyncio.gather(
        asyncio.to_thread(blocking_io),
        asyncio.sleep(1))

    print(f"finished main at {time.strftime('%X')}")


asyncio.run(main())

# Expected output:
#
# started main at 19:50:53
# start blocking_io at 19:50:53
# blocking_io complete at 19:50:54
# finished main at 19:50:54
```

Appeler directement `blocking_io()` dans n'importe quelle coroutine bloquerait la boucle d'événements pendant sa durée, ce qui entraînerait une seconde supplémentaire de temps d'exécution. Au lieu de cela, en utilisant `asyncio.to_thread()`, nous pouvons l'exécuter dans un fil d'exécution séparé sans bloquer la boucle d'événements.

> **Note**
>
> en raison du *GIL*, `asyncio.to_thread()` ne peut généralement être utilisée que pour rendre les fonctions

faisant beaucoup d'entrées-sorties non bloquantes. Cependant, pour les modules d'extension qui relâchent le GIL ou les implémentations Python alternatives qui n'en ont pas, `asyncio.to_thread()` peut également être utilisée pour les fonctions qui sollicitent beaucoup le processeur.

Ajouté dans la version 3.9.

Planification depuis d'autres fils d'exécution

`asyncio.`**`run_coroutine_threadsafe`**(*coro*, *loop*)

Enregistre une coroutine dans la boucle d'exécution actuelle. Cette opération est compatible avec les programmes à multiples fils d'exécution (*thread-safe*).

Renvoie un `concurrent.futures.Future` pour attendre le résultat d'un autre fil d'exécution du système d'exploitation.

Cette fonction est faite pour être appelée par un fil d'exécution distinct de celui dans laquelle la boucle d'événement s'exécute. Exemple :

```
def in_thread(loop: asyncio.AbstractEventLoop) -> None:
    # Run some blocking IO
    pathlib.Path("example.txt").write_text("hello world", encoding="utf8")

    # Create a coroutine
    coro = asyncio.sleep(1, result=3)

    # Submit the coroutine to a given loop
    future = asyncio.run_coroutine_threadsafe(coro, loop)

    # Wait for the result with an optional timeout argument
    assert future.result(timeout=2) == 3

async def amain() -> None:
    # Get the running loop
    loop = asyncio.get_running_loop()

    # Run something in a thread
    await asyncio.to_thread(in_thread, loop)
```

It's also possible to run the other way around. Example :

```
@contextlib.contextmanager
def loop_in_thread() -> Generator[asyncio.AbstractEventLoop]:
    loop_fut = concurrent.futures.Future[asyncio.AbstractEventLoop]()
    stop_event = asyncio.Event()

    async def main() -> None:
        loop_fut.set_result(asyncio.get_running_loop())
        await stop_event.wait()

    with concurrent.futures.ThreadPoolExecutor(1) as tpe:
        complete_fut = tpe.submit(asyncio.run, main())
        for fut in concurrent.futures.as_completed((loop_fut, complete_fut)):
            if fut is loop_fut:
                loop = loop_fut.result()
                try:
                    yield loop
                finally:
                    loop.call_soon_threadsafe(stop_event.set)
```

(suite sur la page suivante)

(suite de la page précédente)

```
            else:
                fut.result()

# Create a loop in another thread
with loop_in_thread() as loop:
    # Create a coroutine
    coro = asyncio.sleep(1, result=3)

    # Submit the coroutine to a given loop
    future = asyncio.run_coroutine_threadsafe(coro, loop)

    # Wait for the result with an optional timeout argument
    assert future.result(timeout=2) == 3
```

Si une exception est levée dans une coroutine, le futur renvoyé en sera averti. Elle peut également être utilisée pour annuler la tâche de la boucle d'événement :

```
try:
    result = future.result(timeout)
except TimeoutError:
    print('The coroutine took too long, cancelling the task...')
    future.cancel()
except Exception as exc:
    print(f'The coroutine raised an exception: {exc!r}')
else:
    print(f'The coroutine returned: {result!r}')
```

Voir la section *exécution concurrente et multi-fils d'exécution* de la documentation.

À la différence des autres fonctions d'*asyncio*, cette fonction requiert que *loop* soit passé de manière explicite.

Ajouté dans la version 3.5.1.

Introspection

`asyncio.current_task(loop=None)`

Renvoie l'instance de la `Task` en cours d'exécution, ou `None` s'il n'y a pas de tâche en cours.

Si *loop* vaut `None`, `get_running_loop()` est appelée pour récupérer la boucle en cours d'exécution.

Ajouté dans la version 3.7.

`asyncio.all_tasks(loop=None)`

Renvoie l'ensemble des `Task` non terminés en cours d'exécution dans la boucle.

Si *loop* vaut `None`, `get_running_loop()` est appelée pour récupérer la boucle en cours d'exécution.

Ajouté dans la version 3.7.

`asyncio.iscoroutine(obj)`

Renvoie `True` si *obj* est un objet coroutine.

Ajouté dans la version 3.4.

Objets *Task*

class `asyncio.Task(coro, *, loop=None, name=None, context=None, eager_start=False)`

Objet compatible avec `Future` qui exécute une *coroutine* Python. Cet objet n'est pas utilisable dans des programmes à fils d'exécution multiples.

Les tâches servent à exécuter des coroutines dans des boucles d'événements. Si une coroutine attend un futur, la tâche interrompt son exécution et attend la fin de ce *futur*. Quand celui-ci est terminé, l'exécution de la

coroutine encapsulée reprend.

Les boucles d'événement fonctionnent de manière *coopérative* : une boucle d'événement exécute une tâche à la fois. Quand une tâche attend la fin d'un futur, la boucle d'événement exécute d'autres tâches, des fonctions de rappel, ou effectue des opérations d'entrées-sorties.

La fonction de haut niveau `asyncio.create_task()` et les fonctions de bas-niveau `loop.create_task()` ou `ensure_future()` permettent de créer des tâches. Il est déconseillé d'instancier manuellement des objets *Task*.

La méthode `cancel()` d'une tâche en cours d'exécution permet d'annuler celle-ci. L'appel de cette méthode force la tâche à lever l'exception `CancelledError` dans la coroutine encapsulée. Si la coroutine attendait un *futur* au moment de l'annulation, celui-ci est annulé.

La méthode `cancelled()` permet de vérifier si la tâche a été annulée. Elle renvoie `True` si la coroutine encapsulée n'a pas ignoré l'exception `CancelledError` et a bien été annulée.

`asyncio.Task` hérite de `Future`, de toute son API, à l'exception de `Future.set_result()` et de `Future.set_exception()`.

An optional keyword-only *context* argument allows specifying a custom `contextvars.Context` for the *coro* to run in. If no *context* is provided, the Task copies the current context and later runs its coroutine in the copied context.

An optional keyword-only *eager_start* argument allows eagerly starting the execution of the `asyncio.Task` at task creation time. If set to `True` and the event loop is running, the task will start executing the coroutine immediately, until the first time the coroutine blocks. If the coroutine returns or raises without blocking, the task will be finished eagerly and will skip scheduling to the event loop.

Modifié dans la version 3.7 : ajout du support du module `contextvars`.

Modifié dans la version 3.8 : ajout du paramètre *name*.

Obsolète depuis la version 3.10 : un avertissement d'obsolescence est émis si *loop* n'est pas spécifié et qu'il n'y a pas de boucle d'événement en cours d'exécution.

Modifié dans la version 3.11 : ajout du paramètre *context*.

Modifié dans la version 3.12 : Added the *eager_start* parameter.

done()

Renvoie `True` si la tâche est *achevée*.

Une tâche est dite *achevée* quand la coroutine encapsulée a soit renvoyé une valeur, soit levé une exception, ou que la tâche a été annulée.

result()

Renvoie le résultat de la tâche.

Si la tâche est *achevée*, le résultat de la coroutine encapsulée est renvoyé (sinon, dans le cas où la coroutine a levé une exception, cette exception est de nouveau levée).

Si la tâche a été *annulée*, cette méthode lève une exception `CancelledError`.

If the Task's result isn't yet available, this method raises an `InvalidStateError` exception.

exception()

Renvoie l'exception de la tâche.

Si la coroutine encapsulée lève une exception, cette exception est renvoyée. Si la coroutine s'est exécutée normalement, cette méthode renvoie `None`.

Si la tâche a été *annulée*, cette méthode lève une exception `CancelledError`.

Si la tâche n'est pas encore *achevée*, cette méthode lève une exception `InvalidStateError`.

add_done_callback(*callback*, *, *context=None*)

Ajoute une fonction de rappel qui sera exécutée quand la tâche sera *achevée*.

Cette méthode ne doit être utilisée que dans du code basé sur les fonctions de rappel de bas-niveau.

Se référer à la documentation de `Future.add_done_callback()` pour plus de détails.

remove_done_callback(*callback*)

Retire *callback* de la liste de fonctions de rappel.

Cette méthode ne doit être utilisée que dans du code basé sur les fonctions de rappel de bas-niveau.

Se référer à la documentation de `Future.remove_done_callback()` pour plus de détails.

get_stack(*, *limit=None*)

Renvoie une liste représentant la pile d'appels de la tâche.

Si la coroutine encapsulée n'est pas terminée, cette fonction renvoie la pile d'appels à partir de l'endroit où celle-ci est interrompue. Si la coroutine s'est terminée normalement ou a été annulée, cette fonction renvoie une liste vide. Si la coroutine a été terminée par une exception, ceci renvoie la pile d'erreurs.

La pile est toujours affichée de l'appelant à l'appelé.

Une seule ligne est renvoyée si la coroutine est suspendue.

L'argument facultatif *limit* définit le nombre maximal d'appels à renvoyer ; par défaut, tous sont renvoyés. L'ordre de la liste diffère selon la nature de celle-ci : les appels les plus récents d'une pile d'appels sont renvoyés, si la pile est une pile d'erreurs, ce sont les appels les plus anciens qui le sont (dans un souci de cohérence avec le module *traceback*).

print_stack(*, *limit=None*, *file=None*)

Affiche la pile d'appels ou d'erreurs de la tâche.

Le format de sortie des appels produits par `get_stack()` est similaire à celui du module *traceback*.

Le paramètre *limit* est directement passé à `get_stack()`.

Le paramètre *file* est un flux d'entrées-sorties sur lequel le résultat est écrit ; par défaut, `sys.stdout`.

get_coro()

Renvoie l'objet *coroutine* encapsulé par la `Task`.

> **Note**
>
> This will return `None` for Tasks which have already completed eagerly. See the *Eager Task Factory*.

Ajouté dans la version 3.8.

Modifié dans la version 3.12 : Newly added eager task execution means result may be `None`.

get_context()

Return the `contextvars.Context` object associated with the task.

Ajouté dans la version 3.12.

get_name()

Renvoie le nom de la tâche.

Si aucun nom n'a été explicitement assigné à la tâche, l'implémentation par défaut d'une *Task asyncio* génère un nom par défaut durant l'instanciation.

Ajouté dans la version 3.8.

set_name(*value*)

Définit le nom de la tâche.

L'argument *value* peut être n'importe quel objet qui sera ensuite converti en chaine de caractères.

Dans l'implémentation par défaut de *Task*, le nom sera visible dans le résultat de `repr()` d'un objet *Task*.

Ajouté dans la version 3.8.

cancel(*msg=None*)

Demande l'annulation d'une tâche.

Provisionne la levée de l'exception `CancelledError` dans la coroutine encapsulée. L'exception sera levée au prochain cycle de la boucle d'exécution.

La coroutine peut alors faire le ménage ou même ignorer la requête en supprimant l'exception à l'aide d'un bloc `try` … … `except CancelledError` … `finally`. Par conséquent, contrairement à `Future.cancel()`, `Task.cancel()` ne garantit pas que la tâche sera annulée, bien qu'ignorer totalement une annulation ne soit ni une pratique courante, ni encouragé. Si la coroutine décide néanmoins de supprimer l'annulation, elle doit appeler `Task.uncancel()` en plus d'intercepter l'exception.

Modifié dans la version 3.9 : ajout du paramètre *msg*.

Modifié dans la version 3.11 : le paramètre `msg` est propagé de la tâche annulée vers celle qui l'attend.

L'exemple ci-dessous illustre comment une coroutine peut intercepter une requête d'annulation :

```
async def cancel_me():
    print('cancel_me(): before sleep')

    try:
        # Wait for 1 hour
        await asyncio.sleep(3600)
    except asyncio.CancelledError:
        print('cancel_me(): cancel sleep')
        raise
    finally:
        print('cancel_me(): after sleep')

async def main():
    # Create a "cancel_me" Task
    task = asyncio.create_task(cancel_me())

    # Wait for 1 second
    await asyncio.sleep(1)

    task.cancel()
    try:
        await task
    except asyncio.CancelledError:
        print("main(): cancel_me is cancelled now")

asyncio.run(main())

# Expected output:
#
#     cancel_me(): before sleep
#     cancel_me(): cancel sleep
#     cancel_me(): after sleep
#     main(): cancel_me is cancelled now
```

cancelled()

Renvoie `True` si la tâche est *annulée*.

La tâche est *annulée* quand l'annulation a été demandée avec `cancel()` et la coroutine encapsulée a propagé l'exception `CancelledError` qui a été levée en son sein.

uncancel()

Décrémente le nombre de demandes d'annulation pour cette tâche.

Renvoie le nombre restant de demandes d'annulation.

Notez qu'une fois l'exécution d'une tâche annulée terminée, les appels ultérieurs à `uncancel()` ne font rien.

Ajouté dans la version 3.11.

Cette méthode est utilisée par les composants internes d'*asyncio* et elle ne devrait pas être utilisée par le code de l'utilisateur final. En particulier, si une tâche est annulée avec succès, cela permet aux structures permettant le multi-fils tels que *Groupes de tâches* et `asyncio.timeout()` de continuer à s'exécuter, isolant l'annulation au bloc concerné. Par exemple :

```
async def make_request_with_timeout():
    try:
        async with asyncio.timeout(1):
            # Structured block affected by the timeout:
            await make_request()
            await make_another_request()
    except TimeoutError:
        log("There was a timeout")
    # Outer code not affected by the timeout:
    await unrelated_code()
```

Alors que le bloc avec `make_request()` et `make_another_request()` peut être annulé en raison du délai d'attente, `unrelated_code()` devrait continuer à s'exécuter même en cas d'atteinte du délai maximal. Ceci est implémenté avec `uncancel()`. Les gestionnaires de contexte `TaskGroup` utilisent `uncancel()` de la même manière.

If end-user code is, for some reason, suppressing cancellation by catching `CancelledError`, it needs to call this method to remove the cancellation state.

When this method decrements the cancellation count to zero, the method checks if a previous `cancel()` call had arranged for `CancelledError` to be thrown into the task. If it hasn't been thrown yet, that arrangement will be rescinded (by resetting the internal `_must_cancel` flag).

Modifié dans la version 3.13 : Changed to rescind pending cancellation requests upon reaching zero.

cancelling()

Renvoie le nombre de demandes d'annulation en attente à cette tâche, c'est-à-dire le nombre d'appels à `cancel()` moins le nombre d'appels à `uncancel()`.

Notez que si ce nombre est supérieur à zéro mais que la tâche est toujours en cours d'exécution, `cancelled()` renvoie toujours `False`. En effet, ce nombre peut être réduit en appelant `uncancel()`, ce qui peut empêcher en fin de compte la tâche d'être annulée si les demandes d'annulation tombent à zéro.

Cette méthode est utilisée par les composants internes d'*asyncio* et ne devrait pas être utilisée par le code de l'utilisateur final. Voir `uncancel()` pour plus de détails.

Ajouté dans la version 3.11.

19.1.3 Flux (*streams*)

Code source : Lib/asyncore.py

Les flux sont des primitives de haut niveau compatibles avec *async/await* pour utiliser les connexions réseau. Les flux permettent d'envoyer et de recevoir des données sans utiliser de fonctions de rappel ou des protocoles de bas niveau.

Voici un exemple de client « écho TCP » écrit en utilisant les flux *asyncio* :

```
import asyncio

async def tcp_echo_client(message):
```

(suite sur la page suivante)

(suite de la page précédente)

```
    reader, writer = await asyncio.open_connection(
        '127.0.0.1', 8888)

    print(f'Send: {message!r}')
    writer.write(message.encode())
    await writer.drain()

    data = await reader.read(100)
    print(f'Received: {data.decode()!r}')

    print('Close the connection')
    writer.close()
    await writer.wait_closed()

asyncio.run(tcp_echo_client('Hello World!'))
```

Voir également la section *Exemples* ci-dessous.

Fonctions de flux

Les fonctions *asyncio* de haut niveau suivantes peuvent être utilisées pour créer et utiliser des flux :

coroutine asyncio.**open_connection**(*host=None*, *port=None*, ***, *limit=None*, *ssl=None*, *family=0*, *proto=0*, *flags=0*, *sock=None*, *local_addr=None*, *server_hostname=None*, *ssl_handshake_timeout=None*, *ssl_shutdown_timeout=None*, *happy_eyeballs_delay=None*, *interleave=None*)

Établit une connexion réseau et renvoie une paire d'objets `(lecteur, écrivain)`.

Les objets *lecteur* et *écrivain* renvoyés sont des instances des classes `StreamReader` et `StreamWriter`.

limit détermine la limite de taille de tampon utilisée par l'instance `StreamReader` renvoyée. Par défaut, *limit* est fixée à 64 Kio.

Le reste des arguments est passé directement à `loop.create_connection()`.

> **Note**
>
> l'argument *sock* transfère la propriété du connecteur réseau au `StreamWriter` créé. Pour fermer le connecteur, appelez sa méthode `close()`.

Modifié dans la version 3.7 : ajout du paramètre *ssl_handshake_timeout*.

Modifié dans la version 3.8 : Added the *happy_eyeballs_delay* and *interleave* parameters.

Modifié dans la version 3.10 : suppression du paramètre *loop*.

Modifié dans la version 3.11 : ajout du paramètre *ssl_shutdown_timeout*.

coroutine asyncio.**start_server**(*client_connected_cb*, *host=None*, *port=None*, ***, *limit=None*, *family=socket.AF_UNSPEC*, *flags=socket.AI_PASSIVE*, *sock=None*, *backlog=100*, *ssl=None*, *reuse_address=None*, *reuse_port=None*, *keep_alive=None*, *ssl_handshake_timeout=None*, *ssl_shutdown_timeout=None*, *start_serving=True*)

Démarre un serveur de connexions

La fonction de rappel *client_connected_cb* est appelée chaque fois qu'une nouvelle connexion client est établie. Elle reçoit une paire d'arguments `(lecteur, écrivain)`, instances des classes `StreamReader` et `StreamWriter`.

client_connected_cb peut être un simple appelable ou une fonction *coroutine* ; s'il s'agit d'une fonction coroutine,

elle sera automatiquement planifiée en tant que `Task`.

limit détermine la limite de taille de tampon utilisée par l'instance `StreamReader` renvoyée. Par défaut, *limit* est fixée à 64 Kio.

Le reste des arguments est passé directement à `loop.create_server()`.

> **Note**
>
> l'argument *sock* transfère la propriété du connecteur au serveur créé. Pour fermer le connecteur, appelez la méthode `close()` du serveur.

Modifié dans la version 3.7 : ajout des paramètres *ssl_handshake_timeout* et *start_serving*.

Modifié dans la version 3.10 : suppression du paramètre *loop*.

Modifié dans la version 3.11 : ajout du paramètre *ssl_shutdown_timeout*.

Modifié dans la version 3.13 : Added the *keep_alive* parameter.

Connecteurs Unix (*sockets*)

coroutine asyncio.**open_unix_connection**(*path=None*, *, *limit=None*, *ssl=None*, *sock=None*, *server_hostname=None*, *ssl_handshake_timeout=None*, *ssl_shutdown_timeout=None*)

Ouvre un connecteur Unix et renvoie une paire de `(lecteur, écrivain)`.

Similaire à `open_connection()` mais fonctionne sur les connecteurs Unix.

Voir aussi la documentation de `loop.create_unix_connection()`.

> **Note**
>
> l'argument *sock* transfère la propriété du connecteur réseau au `StreamWriter` créé. Pour fermer le connecteur, appelez sa méthode `close()`.

Availability : Unix.

Modifié dans la version 3.7 : ajout du paramètre *ssl_handshake_timeout*. Le paramètre *path* peut désormais être un *objet simili-chemin*

Modifié dans la version 3.10 : suppression du paramètre *loop*.

Modifié dans la version 3.11 : ajout du paramètre *ssl_shutdown_timeout*.

coroutine asyncio.**start_unix_server**(*client_connected_cb*, *path=None*, *, *limit=None*, *sock=None*, *backlog=100*, *ssl=None*, *ssl_handshake_timeout=None*, *ssl_shutdown_timeout=None*, *start_serving=True*)

Démarre un connecteur Unix en mode serveur.

Similaire à `start_server()` mais fonctionne avec les connecteurs Unix.

Voir aussi la documentation de `loop.create_unix_server()`.

> **Note**
>
> l'argument *sock* transfère la propriété du connecteur au serveur créé. Pour fermer le connecteur, appelez la méthode `close()` du serveur.

Availability : Unix.

Modifié dans la version 3.7 : ajout des paramètres *ssl_handshake_timeout* et *start_serving*. Le paramètre *chemin*

peut désormais être un *simili-chemin*.

Modifié dans la version 3.10 : suppression du paramètre *loop*.

Modifié dans la version 3.11 : ajout du paramètre *ssl_shutdown_timeout*.

Flux lecteurs (*StreamReader*)

class asyncio.**StreamReader**

Représente un objet lecteur qui fournit des API pour lire les données du flux d'entrée-sortie. En tant que *itérable asynchrone*, l'objet prend en charge l'instruction `async for`.

Il n'est pas recommandé d'instancier directement les objets *StreamReader* ; utilisez `open_connection()` et `start_server()` à la place.

feed_eof()

Acknowledge the EOF.

coroutine read(*n=-1*)

Lit jusqu'à *n* octets du flux.

Si *n* n'est pas fourni ou défini à `-1`, lit jusqu'à *EOF*, puis renvoie tous les `bytes` lus. Si *EOF* a été reçu et que le tampon interne est vide, renvoie un objet `bytes` vide.

Si *n* vaut `0`, renvoie immédiatement un objet `bytes` vide.

Si *n* est positif, renvoie au plus *n* `bytes` disponibles dès qu'au moins 1 octet est disponible dans le tampon interne. Si *EOF* est reçu avant qu'aucun octet ne soit lu, renvoie un objet `bytes` vide.

coroutine readline()

Lit une ligne, où une « ligne » est une séquence d'octets se terminant par `\n`.

Si *EOF* est reçu et `\n` n'a pas été trouvé, la méthode renvoie des données partiellement lues.

Si *EOF* est reçu et que le tampon interne est vide, renvoie un objet `bytes` vide.

coroutine readexactly(*n*)

Lit exactement *n* octets.

Lève une `IncompleteReadError` si *EOF* est atteint avant que *n* octets ne puissent être lus. Utilisez l'attribut `IncompleteReadError.partial` pour obtenir les données partiellement lues.

coroutine readuntil(*separator=b'\n'*)

Lit les données du flux jusqu'à ce que le *separator* soit trouvé.

En cas de succès, les données et le séparateur sont supprimés du tampon interne (consommés). Les données renvoyées incluent le séparateur à la fin.

Si la quantité de données lues dépasse la limite de flux configurée, une exception `LimitOverrunError` est levée et les données sont laissées dans le tampon interne et peuvent être lues à nouveau.

Si *EOF* est atteint avant que le séparateur complet ne soit trouvé, une exception `IncompleteReadError` est levée et le tampon interne est réinitialisé. L'attribut `IncompleteReadError.partial` peut contenir une partie du séparateur.

The *separator* may also be a tuple of separators. In this case the return value will be the shortest possible that has any separator as the suffix. For the purposes of `LimitOverrunError`, the shortest possible separator is considered to be the one that matched.

Ajouté dans la version 3.5.2.

Modifié dans la version 3.13 : The *separator* parameter may now be a `tuple` of separators.

at_eof()

Renvoie `True` si le tampon est vide et que `feed_eof()` a été appelée.

Flux écrivains (*StreamWriter*)

class asyncio.**StreamWriter**

Représente un objet écrivain qui fournit des API pour écrire des données dans le flux d'entrée-sortie.

Il n'est pas recommandé d'instancier directement les objets *StreamWriter* ; utilisez `open_connection()` et `start_server()` à la place.

write(*data*)

La méthode tente d'écrire immédiatement les *data* dans le connecteur sous-jacent. Si cela échoue, les données sont mises en file d'attente dans un tampon d'écriture interne jusqu'à ce qu'elles puissent être envoyées.

La méthode doit être utilisée avec la méthode `drain()` :

```
stream.write(data)
await stream.drain()
```

writelines(*data*)

La méthode écrit immédiatement une liste (ou tout itérable) d'octets dans le connecteur sous-jacent. Si cela échoue, les données sont mises en file d'attente dans un tampon d'écriture interne jusqu'à ce qu'elles puissent être envoyées.

La méthode doit être utilisée avec la méthode `drain()` :

```
stream.writelines(lines)
await stream.drain()
```

close()

La méthode ferme le flux et le connecteur sous-jacent.

La méthode doit être utilisée, bien que ce ne soit pas obligatoire, avec la méthode `wait_closed()` :

```
stream.close()
await stream.wait_closed()
```

can_write_eof()

Renvoie `True` si le transport sous-jacent gère la méthode `write_eof()`, `False` sinon.

write_eof()

Ferme le flux en écriture après le vidage des données d'écriture en mémoire tampon.

transport

Renvoie le transport asynchrone sous-jacent.

get_extra_info(*name*, *default=None*)

Donne accès aux informations de transport facultatives ; voir `BaseTransport.get_extra_info()` pour plus de détails.

coroutine drain()

Attend qu'il soit approprié de reprendre l'écriture dans le flux. Par exemple :

```
writer.write(data)
await writer.drain()
```

Il s'agit d'une méthode de contrôle de flux qui interagit avec le tampon d'écriture entrée-sortie sous-jacent. Lorsque la taille du tampon atteint la limite haute, *drain()* bloque jusqu'à ce que la taille du tampon soit drainée jusqu'à la limite basse et que l'écriture puisse reprendre. Lorsqu'il n'y a rien à attendre, `drain()` termine immédiatement.

coroutine start_tls(*sslcontext*, *, *server_hostname=None*, *ssl_handshake_timeout=None*, *ssl_shutdown_timeout=None*)

Bascule la connexion basée sur le flux existant vers TLS.

Paramètres :
— *sslcontext* : une instance configurée de `SSLContext`.
— *server_hostname* : définit ou remplace le nom d'hôte auquel le certificat du serveur cible sera comparé.
— *ssl_handshake_timeout* est le temps en secondes à attendre pour que la poignée de main TLS se termine avant d'abandonner la connexion. `60.0` secondes si `None` (par défaut).
— *ssl_shutdown_timeout* is the time in seconds to wait for the SSL shutdown to complete before aborting the connection. `30.0` seconds if `None` (default).

Ajouté dans la version 3.11.

Modifié dans la version 3.12 : ajout du paramètre *ssl_shutdown_timeout*.

is_closing()

Renvoie `True` si le flux est fermé ou en cours de fermeture.

Ajouté dans la version 3.7.

coroutine wait_closed()

Attend que le flux soit fermé.

Doit être appelée après `close()` pour attendre que la connexion sous-jacente soit fermée, en s'assurant que toutes les données ont été vidées avant, par exemple, de quitter le programme.

Ajouté dans la version 3.7.

Exemples

Client d'écho TCP utilisant des flux

Client d'écho TCP utilisant la fonction `asyncio.open_connection()` :

```
import asyncio

async def tcp_echo_client(message):
    reader, writer = await asyncio.open_connection(
        '127.0.0.1', 8888)

    print(f'Send: {message!r}')
    writer.write(message.encode())
    await writer.drain()

    data = await reader.read(100)
    print(f'Received: {data.decode()!r}')

    print('Close the connection')
    writer.close()
    await writer.wait_closed()

asyncio.run(tcp_echo_client('Hello World!'))
```

Voir aussi

L'exemple *Client écho en TCP* utilise la méthode de bas niveau `loop.create_connection()`.

Serveur d'écho TCP utilisant des flux

Serveur d'écho TCP utilisant la fonction `asyncio.start_server()` :

```
import asyncio
```

(suite sur la page suivante)

(suite de la page précédente)

```
async def handle_echo(reader, writer):
    data = await reader.read(100)
    message = data.decode()
    addr = writer.get_extra_info('peername')

    print(f"Received {message!r} from {addr!r}")

    print(f"Send: {message!r}")
    writer.write(data)
    await writer.drain()

    print("Close the connection")
    writer.close()
    await writer.wait_closed()

async def main():
    server = await asyncio.start_server(
        handle_echo, '127.0.0.1', 8888)

    addrs = ', '.join(str(sock.getsockname()) for sock in server.sockets)
    print(f'Serving on {addrs}')

    async with server:
        await server.serve_forever()

asyncio.run(main())
```

Voir aussi

L'exemple de *Serveur écho en TCP* utilise la méthode `loop.create_server()`.

Récupération des en-têtes HTTP

Exemple simple d'interrogation des en-têtes HTTP de l'URL transmise sur la ligne de commande :

```
import asyncio
import urllib.parse
import sys

async def print_http_headers(url):
    url = urllib.parse.urlsplit(url)
    if url.scheme == 'https':
        reader, writer = await asyncio.open_connection(
            url.hostname, 443, ssl=True)
    else:
        reader, writer = await asyncio.open_connection(
            url.hostname, 80)

    query = (
        f"HEAD {url.path or '/'} HTTP/1.0\r\n"
        f"Host: {url.hostname}\r\n"
        f"\r\n"
    )

    writer.write(query.encode('latin-1'))
```

(suite sur la page suivante)

(suite de la page précédente)

```
    while True:
        line = await reader.readline()
        if not line:
            break

        line = line.decode('latin1').rstrip()
        if line:
            print(f'HTTP header> {line}')

    # Ignore the body, close the socket
    writer.close()
    await writer.wait_closed()

url = sys.argv[1]
asyncio.run(print_http_headers(url))
```

Utilisation :

```
python example.py http://example.com/path/page.html
```

ou avec HTTPS :

```
python example.py https://example.com/path/page.html
```

Ouverture d'un connecteur pour attendre les données à l'aide de flux

Coroutine attendant qu'un connecteur reçoive des données en utilisant la fonction `open_connection()` :

```
import asyncio
import socket

async def wait_for_data():
    # Get a reference to the current event loop because
    # we want to access low-level APIs.
    loop = asyncio.get_running_loop()

    # Create a pair of connected sockets.
    rsock, wsock = socket.socketpair()

    # Register the open socket to wait for data.
    reader, writer = await asyncio.open_connection(sock=rsock)

    # Simulate the reception of data from the network
    loop.call_soon(wsock.send, 'abc'.encode())

    # Wait for data
    data = await reader.read(100)

    # Got data, we are done: close the socket
    print("Received:", data.decode())
    writer.close()
    await writer.wait_closed()

    # Close the second socket
    wsock.close()

asyncio.run(wait_for_data())
```

Voir aussi

L'exemple *Connexion de connecteurs existants* utilise un protocole de bas niveau et la méthode `loop.create_connection()`.

L'exemple *Surveillance des événements de lecture pour un descripteur de fichier* utilise la méthode de bas niveau `loop.add_reader()` pour surveiller un descripteur de fichier.

19.1.4 Primitives de synchronisation

Code source : Lib/asyncio/locks.py

Les primitives de synchronisation *asyncio* sont conçues pour être similaires à celles du module `threading` avec deux mises en garde importantes :

- les primitives *asyncio* ne sont pas *thread-safe*, elles ne doivent donc pas être utilisées pour la synchronisation des fils d'exécution du système d'exploitation (utilisez `threading` pour cela) ;
- les méthodes de ces primitives de synchronisation n'acceptent pas l'argument *timeout* ; utilisez la fonction `asyncio.wait_for()` pour effectuer des opérations avec des délais d'attente.

asyncio possède les primitives de synchronisation de base suivantes :

- `Lock`
- `Event`
- `Condition`
- `Semaphore`
- `BoundedSemaphore`
- `Barrier`

Verrou (*lock*)

class asyncio.**Lock**

Implémente un verrou exclusif (*mutex*) pour les tâches asynchrones. Ce n'est pas compatible avec les programmes à fils d'exécution multiples.

Un verrou *asyncio* peut être utilisé pour garantir un accès exclusif à une ressource partagée.

La meilleure façon d'utiliser un verrou est une instruction `async with`

```
lock = asyncio.Lock()

# ... later
async with lock:
    # access shared state
```

ce qui équivaut à :

```
lock = asyncio.Lock()

# ... later
await lock.acquire()
try:
    # access shared state
finally:
    lock.release()
```

Modifié dans la version 3.10 : suppression du paramètre *loop*.

coroutine acquire()

Verrouille (ou acquiert) le verrou.

Cette méthode attend que le verrou soit déverrouillé (*unlocked*), le verrouille (positionné sur *locked*) et renvoie `True`.

Lorsque plus d'une coroutine est bloquée dans `acquire()` en attendant que le verrou soit déverrouillé, seule une coroutine continue finalement.

L'acquisition d'un verrou est *équitable* : la coroutine qui acquiert le verrou est celle qui était la première à attendre le verrou.

release()

Libère le verrou.

Lorsque le verrou est verrouillé, le déverrouille et termine.

Si le verrou est déjà déverrouillé, une `RuntimeError` est levée.

locked()

Renvoie `True` si le verrou est verrouillé.

Événement (*Event*)

class asyncio.**Event**

Objet événement. Non compatible avec les programmes à plusieurs fils d'exécution.

Un événement asynchrone peut être utilisé pour notifier plusieurs tâches asynchrones qu'un événement s'est produit.

Un objet *Event* gère un drapeau interne qui peut être activé (ou mis à *vrai*) avec la méthode `set()` et désactivé (ou mis à *faux*) avec la méthode `clear()`. La méthode `wait()` se bloque jusqu'à ce que l'indicateur soit activé. L'indicateur est initialement désactivé.

Modifié dans la version 3.10 : suppression du paramètre *loop*. Exemple :

```
async def waiter(event):
    print('waiting for it ...')
    await event.wait()
    print('... got it!')

async def main():
    # Create an Event object.
    event = asyncio.Event()

    # Spawn a Task to wait until 'event' is set.
    waiter_task = asyncio.create_task(waiter(event))

    # Sleep for 1 second and set the event.
    await asyncio.sleep(1)
    event.set()

    # Wait until the waiter task is finished.
    await waiter_task

asyncio.run(main())
```

coroutine wait()

Attend que l'évènement soit activé.

Si l'événement est activé (*vrai*), renvoie `True` immédiatement. Sinon bloque jusqu'à ce qu'une autre tâche appelle `set()`.

set()

Active l'événement.

Toutes les tâches en attente de l'événement sont immédiatement réveillées.

clear()

Efface (désactive) l'événement.

Les tâches en attente sur `wait()` seront désormais bloquées jusqu'à ce que la méthode `set()` soit à nouveau appelée.

is_set()

Renvoie `True` si l'évènement est actif.

Condition

class asyncio.**Condition**(*lock=None*)

Objet Condition. Non compatible avec les programmes à plusieurs fils d'exécution.

Une primitive de condition asynchrone peut être utilisée par une tâche pour attendre qu'un événement se produise, puis obtenir un accès exclusif à une ressource partagée.

Essentiellement, un objet *Condition* combine les fonctionnalités d'un `Event` et d'un `Lock`. Il est possible que plusieurs objets *Condition* partagent un seul verrou, ce qui permet de coordonner l'accès exclusif à une ressource partagée entre différentes tâches intéressées par des états particuliers de cette ressource partagée.

L'argument optionnel *lock* doit être un objet `Lock` ou `None`. Dans ce dernier cas, un nouvel objet *Lock* est créé automatiquement.

Modifié dans la version 3.10 : suppression du paramètre *loop*.

La meilleure façon d'utiliser une *Condition* est une instruction `async with`

```
cond = asyncio.Condition()

# ... later
async with cond:
    await cond.wait()
```

ce qui équivaut à :

```
cond = asyncio.Condition()

# ... later
await cond.acquire()
try:
    await cond.wait()
finally:
    cond.release()
```

coroutine acquire()

Verrouille le verrou sous-jacent.

Cette méthode attend que le verrou sous-jacent soit déverrouillé, le verrouille et renvoie `True`.

notify(*n=1*)

Wake up *n* tasks (1 by default) waiting on this condition. If fewer than *n* tasks are waiting they are all awakened.

Le verrou doit être verrouillé avant que cette méthode ne soit appelée et libéré peu de temps après. S'il est appelé avec un verrou déverrouillé, une erreur `RuntimeError` est levée.

locked()

Renvoie `True` si le verrou sous-jacent est verrouillé.

notify_all()

Réveille toutes les tâches en attente sur cette condition.

Cette méthode agit comme `notify()`, mais réveille toutes les tâches en attente.

Le verrou doit être verrouillé avant que cette méthode ne soit appelée et libéré peu de temps après. S'il est appelé avec un verrou déverrouillé, une erreur `RuntimeError` est levée.

release()

Libère le verrou sous-jacent.

Lorsqu'elle est invoquée sur un verrou déverrouillé, une `RuntimeError` est levée.

coroutine wait()

Attend d'être notifié.

Si la tâche appelante n'a pas verrouillé le verrou lorsque cette méthode est appelée, une `RuntimeError` est levée.

Cette méthode libère le verrou sous-jacent, puis se bloque jusqu'à ce qu'elle soit réveillée par un appel `notify()` ou `notify_all()`. Une fois réveillée, la *Condition* verrouille à nouveau son verrou et cette méthode renvoie `True`.

Note that a task *may* return from this call spuriously, which is why the caller should always re-check the state and be prepared to `wait()` again. For this reason, you may prefer to use `wait_for()` instead.

coroutine wait_for(*predicate*)

Attend jusqu'à ce qu'un prédicat devienne vrai.

The predicate must be a callable which result will be interpreted as a boolean value. The method will repeatedly `wait()` until the predicate evaluates to *true*. The final value is the return value.

Sémaphore

class asyncio.**Semaphore**(*value=1*)

Objet *Sémaphore*. Non compatible avec les programmes à plusieurs fils d'exécution.

Un sémaphore gère un compteur interne qui est décrémenté à chaque appel `acquire()` et incrémenté à chaque appel `release()`. Le compteur ne peut jamais descendre en dessous de zéro ; quand `acquire()` trouve qu'il est égal à zéro, il se bloque, en attendant qu'une tâche appelle `release()`.

L'argument optionnel *value* donne la valeur initiale du compteur interne (`1` par défaut). Si la valeur donnée est inférieure à `0` une `ValueError` est levée.

Modifié dans la version 3.10 : suppression du paramètre *loop*.

La meilleure façon d'utiliser un sémaphore est une instruction `async with`

```
sem = asyncio.Semaphore(10)

# ... later
async with sem:
    # work with shared resource
```

ce qui équivaut à :

```
sem = asyncio.Semaphore(10)

# ... later
await sem.acquire()
try:
    # work with shared resource
finally:
    sem.release()
```

coroutine acquire()

Acquiert un sémaphore.

Si le compteur interne est supérieur à zéro, le décrémente d'une unité et renvoie `True` immédiatement. Si c'est zéro, attend que `release()` soit appelée et renvoie `True`.

locked()

Renvoie `True` si le sémaphore ne peut pas être acquis immédiatement.

release()

Relâche un sémaphore, incrémentant le compteur interne d'une unité. Peut réveiller une tâche en attente d'acquisition du sémaphore.

Contrairement à `BoundedSemaphore`, `Semaphore` permet de faire plus d'appels `release()` que d'appels `acquire()`.

Sémaphore capé (*BoundedSemaphore*)

class asyncio.**BoundedSemaphore**(*value=1*)

Objet sémaphore capé. Non compatible avec les programmes à plusieurs fils d'exécution.

Bounded Semaphore est une version de `Semaphore` qui lève une `ValueError` dans `release()` s'il augmente le compteur interne au-dessus de la *value* initiale.

Modifié dans la version 3.10 : suppression du paramètre *loop*.

Barrière (*Barrier*)

class asyncio.**Barrier**(*parties*)

Objet barrière. Non compatible avec les programmes à plusieurs fils d'exécution.

Une barrière est une simple primitive de synchronisation qui permet de bloquer jusqu'à ce que *parties* tâches l'attendent. Les tâches attendent sur la méthode `wait()` et sont bloquées jusqu'à ce que le nombre spécifié de tâches attendent sur `wait()`. À ce stade, toutes les tâches en attente se débloquent simultanément.

`async with` peut être utilisé comme alternative à l'attente sur `wait()`.

La barrière peut être réutilisée un nombre illimité de fois.

Exemple :

```
async def example_barrier():
   # barrier with 3 parties
   b = asyncio.Barrier(3)

   # create 2 new waiting tasks
   asyncio.create_task(b.wait())
   asyncio.create_task(b.wait())

   await asyncio.sleep(0)
   print(b)

   # The third .wait() call passes the barrier
   await b.wait()
   print(b)
   print("barrier passed")

   await asyncio.sleep(0)
   print(b)

asyncio.run(example_barrier())
```

Le résultat de cet exemple est :

```
<asyncio.locks.Barrier object at 0x... [filling, waiters:2/3]>
<asyncio.locks.Barrier object at 0x... [draining, waiters:0/3]>
barrier passed
<asyncio.locks.Barrier object at 0x... [filling, waiters:0/3]>
```

Ajouté dans la version 3.11.

coroutine wait()

Passe la barrière. Lorsque toutes les tâches bloquées à la barrière ont appelé cette fonction, elles sont toutes débloquées simultanément.

Lorsqu'une tâche en attente ou bloquée à la barrière est annulée, cette tâche sort de la barrière qui reste dans le même état. Si la barrière est en cours de « remplissage », le nombre de tâche en attente diminue de 1.

La valeur de retour est un entier compris entre 0 et `parties-1`, différent pour chaque tâche. Cela peut être utilisé pour sélectionner une tâche qui fera du ménage, par exemple :

```
...
async with barrier as position:
   if position == 0:
      # Only one task prints this
      print('End of *draining phase*')
```

Cette méthode peut lever une exception *BrokenBarrierError* si la barrière est brisée ou réinitialisée alors qu'une tâche est en attente. Cela peut lever une *CancelledError* si une tâche est annulée.

coroutine reset()

Ramène la barrière à l'état vide par défaut. Toutes les tâches en attente reçoivent l'exception *BrokenBarrierError*.

Si une barrière est brisée, il peut être préférable de la quitter et d'en créer une nouvelle.

coroutine abort()

Put the barrier into a broken state. This causes any active or future calls to *wait()* to fail with the *BrokenBarrierError*. Use this for example if one of the tasks needs to abort, to avoid infinite waiting tasks.

parties

Le nombre de tâches nécessaires pour franchir la barrière.

n_waiting

Le nombre de tâches actuellement en attente à la barrière pendant le remplissage.

broken

Booléen qui vaut `True` si la barrière est rompue.

exception asyncio.**BrokenBarrierError**

Cette exception, une sous-classe de *RuntimeError*, est déclenchée lorsque l'objet *Barrier* est réinitialisé ou cassé.

Modifié dans la version 3.9 : l'acquisition d'un verrou en utilisant `wait lock` ou `yield from lock` ou `with` (`with await lock`, `with (yield from lock)`) a été supprimée. Utilisez `async with lock` à la place.

19.1.5 Sous-processus

Code source : Lib/asyncio/subprocess.py, Lib/asyncio/base_subprocess.py

---

Cette section décrit des API de haut niveau de *asyncio* pour créer et gérer des sous-processus via `async`/`await`.

Voici un exemple de comment *asyncio* peut lancer une commande shell et obtenir son résultat :

```
import asyncio

async def run(cmd):
    proc = await asyncio.create_subprocess_shell(
        cmd,
        stdout=asyncio.subprocess.PIPE,
        stderr=asyncio.subprocess.PIPE)

    stdout, stderr = await proc.communicate()

    print(f'[{cmd!r} exited with {proc.returncode}]')
    if stdout:
        print(f'[stdout]\n{stdout.decode()}')
    if stderr:
        print(f'[stderr]\n{stderr.decode()}')

asyncio.run(run('ls /zzz'))
```

affiche :

```
['ls /zzz' exited with 1]
[stderr]
ls: /zzz: No such file or directory
```

Puisque toutes les fonctions à sous-processus d'*asyncio* sont synchrones et qu'*asyncio* fournit de nombreux outils pour travailler avec de telles fonctions, il est facile d'exécuter et de surveiller de nombreux processus en parallèle. Il est en effet trivial de modifier l'exemple ci-dessus pour exécuter plusieurs commandes simultanément :

```
async def main():
    await asyncio.gather(
        run('ls /zzz'),
        run('sleep 1; echo "hello"'))

asyncio.run(main())
```

Voir également la section *Exemples*.

Créer des sous-processus

coroutine asyncio.**create_subprocess_exec**(*program*, **args*, *stdin=None*, *stdout=None*, *stderr=None*, *limit=None*, ***kwds*)

Crée un sous-processus.

The *limit* argument sets the buffer limit for `StreamReader` wrappers for `Process.stdout` and `Process.stderr` (if `subprocess.PIPE` is passed to *stdout* and *stderr* arguments).

Renvoie une instance de `Process`.

Voir la documentation de `loop.subprocess_exec()` pour d'autres paramètres.

Modifié dans la version 3.10 : suppression du paramètre *loop*.

coroutine asyncio.**create_subprocess_shell**(*cmd*, *stdin=None*, *stdout=None*, *stderr=None*, *limit=None*, ***kwds*)

Exécute la commande *cmd* dans un *shell*.

The *limit* argument sets the buffer limit for `StreamReader` wrappers for `Process.stdout` and `Process.stderr` (if `subprocess.PIPE` is passed to *stdout* and *stderr* arguments).

Renvoie une instance de `Process`.

Voir la documentation de `loop.subprocess_shell()` pour d'autres paramètres.

Important

Il est de la responsabilité de l'application de s'assurer que tous les espaces et les caractères spéciaux sont correctement mis entre guillemets pour éviter les vulnérabilités de type injection de code. La fonction `shlex.quote()` peut être utilisée pour l'échappement des espaces et caractères spéciaux dans les chaînes utilisées pour construire des commandes shell.

Modifié dans la version 3.10 : suppression du paramètre *loop*.

Note

les sous-processus sont disponibles pour Windows si un `ProactorEventLoop` est utilisé. Voir *Support des sous-processus sous Windows* pour plus de précisions.

Voir aussi

asyncio propose aussi les API de « bas niveau » suivantes pour travailler avec les sous-processus : `loop.subprocess_exec()`, `loop.subprocess_shell()`, `loop.connect_read_pipe()`, `loop.connect_write_pipe()`, ainsi que les *Transports entre sous-processus* et *Protocoles liés aux sous-processus*.

Constantes

`asyncio.subprocess.`**`PIPE`**

Peut être passé aux paramètres *stdin*, *stdout* ou *stderr*.

Si *PIPE* est passé au paramètre *stdin*, l'attribut `Process.stdin` ne pointera pas vers une instance de `StreamWriter`.

Si *PIPE* est passé au paramètre *stdout* ou *stderr*, l'attribut `Process.stdout` et `Process.stderr` pointeront vers des instances de `StreamReader`.

`asyncio.subprocess.`**`STDOUT`**

Une valeur spéciale qui peut être passée au paramètre *stderr* et qui indique que la sortie d'erreur doit être redirigée vers la sortie standard.

`asyncio.subprocess.`**`DEVNULL`**

Une valeur spéciale qui peut être passée à l'argument *stdin*, *stdout* ou *stderr* des fonctions créant des processus. Elle implique l'utilisation du fichier `os.devnull` pour le flux correspondant du processus.

Interagir avec les sous-processus

Les fonctions `create_subprocess_exec()` et `create_subprocess_shell()` renvoient des instances de la classe *Process*. *Process* est une enveloppe de haut niveau qui permet de communiquer avec les sous-processus et de surveiller leur achèvement.

`class` `asyncio.subprocess.`**`Process`**

Objet qui encapsule les processus du système d'exploitation créés par les fonctions `create_subprocess_exec()` et `create_subprocess_shell()`.

Cette classe est conçue pour avoir une API similaire à la classe `subprocess.Popen`, mais il existe quelques différences notables :

— contrairement à *Popen*, les instances Process n'ont pas d'équivalent à la méthode `poll()` ;
— les méthodes `communicate()` et `wait()` n'ont pas de paramètre *timeout* : utilisez la fonction `wait_for()` ;
— `asyncio.subprocess.Process.wait()` est asynchrone, tandis que la méthode `subprocess.Popen.wait()` est implémentée comme une boucle bloquante ;
— le paramètre *universal_newlines* n'est pas pris en charge.

Cette classe n'est *pas conçue pour un contexte multi-fils*.

Voir aussi la section *sous-processus et fils d'exécution*.

coroutine wait()

Attend que le sous processus s'arrête.

Définit et renvoie l'attribut `returncode`.

> **Note**
>
> cette méthode peut générer un interblocage quand `stdout=PIPE` ou `stderr=PIPE` est utilisé et que le sous-processus génère tellement de sorties qu'il se bloque, dans l'attente que le tampon du tube côté OS accepte des données supplémentaires. Pour éviter cette situation, choisissez la méthode `communicate()` quand vous utilisez des tubes.

coroutine communicate(*input=None*)

Interagit avec le processus :

1. envoie des données sur le *stdin* (si *input* n'est pas `None`) ;
2. closes *stdin* ;
3. lit les données sur *stdout* et *stderr*, jusqu'à ce que le EOF soit atteint ;
4. attend que le processus s'arrête.

Le paramètre optionnel *input* (objet de type `bytes`) représente les données transmises au sous-processus.

Renvoie un *n*-uplet `(stdout_data, stderr_data)`.

Si l'une des exceptions `BrokenPipeError` ou `ConnectionResetError` est levée lors de l'écriture de *input* dans *stdin*, l'exception est ignorée. Cette condition se produit lorsque le processus se termine avant que toutes les données ne soient écrites dans *stdin*.

Si vous souhaitez envoyer des données au processus *stdin*, le processus doit être créé avec `stdin=PIPE`. De même, pour obtenir autre chose que `None` dans le *n*-uplet résultat, le processus doit être créé avec les arguments `stdout=PIPE` et/ou `stderr=PIPE`.

Notez que les données lues sont mises en cache en mémoire, donc n'utilisez pas cette méthode si la taille des données est importante voire illimitée.

Modifié dans la version 3.12 : *stdin* gets closed when *input=None* too.

send_signal(*signal*)

Envoie le signal *signal* au sous-processus.

> **Note**
>
> On Windows, `SIGTERM` is an alias for `terminate()`. `CTRL_C_EVENT` and `CTRL_BREAK_EVENT` can be sent to processes started with a *creationflags* parameter which includes `CREATE_NEW_PROCESS_GROUP`.

terminate()

Arrête le sous-processus.

On POSIX systems this method sends `SIGTERM` to the child process.

On Windows the Win32 API function `TerminateProcess()` is called to stop the child process.

kill()

Arrête le sous-processus.

Sur les systèmes POSIX, cette méthode envoie `signal.SIGTERM` au sous-processus.

Sous Windows, cette méthode est un alias pour `terminate()`.

stdin

Flux d'entrée standard (`StreamWriter`) ou `None` si le processus a été créé avec `stdin=None`.

stdout

Flux de sortie standard (`StreamReader`) ou `None` si le processus a été créé avec `stdout=None`.

stderr

Flux d'erreur standard (`StreamReader`) ou `None` si le processus a été créé avec `stderr=None`.

> **Avertissement**
>
> utilisez la méthode *communicate()* plutôt que *process.stdin.write()*, *wait process.stdout.read()* ou *wait process.stderr.read()*. Cela évite les interblocages dus aux flux qui interrompent la lecture ou l'écriture et bloquent le processus enfant.

pid

Numéro d'identification du processus (PID, pour *Process Identification Number* en anglais).

Notez que pour les processus créés par la fonction `create_subprocess_shell()`, cet attribut est le PID du shell généré.

returncode

Code de retour du processus quand il se termine.

Une valeur `None` indique que le processus n'est pas encore terminé.

Une valeur négative `-N` indique que le sous-processus a été terminé par le signal `N` (seulement sur les systèmes *POSIX*).

Sous-processus et fils d'exécution

La boucle d'événement asynchrone standard prend en charge l'exécution de sous-processus à partir de différents fils d'exécution par défaut.

Sous Windows, les sous-processus sont fournis par *ProactorEventLoop* uniquement (par défaut), *SelectorEventLoop* ne prend pas en charge les sous-processus.

Notez que ces implémentations alternatives de la boucle d'événements peuvent comporter leurs propres limitations. Veuillez vous référer à leur documentation.

> **Voir aussi**
>
> la section *Exécution concurrente et multi-fils d'exécution*.

Exemples

Un exemple utilisant la classe *Process* pour contrôler un sous-processus et la classe *StreamReader* pour lire sa sortie standard.

Le sous-processus est créé par la fonction *create_subprocess_exec()* :

```
import asyncio
import sys

async def get_date():
    code = 'import datetime; print(datetime.datetime.now())'

    # Create the subprocess; redirect the standard output
    # into a pipe.
```

(suite sur la page suivante)

(suite de la page précédente)

```
    proc = await asyncio.create_subprocess_exec(
        sys.executable, '-c', code,
        stdout=asyncio.subprocess.PIPE)

    # Read one line of output.
    data = await proc.stdout.readline()
    line = data.decode('ascii').rstrip()

    # Wait for the subprocess exit.
    await proc.wait()
    return line

date = asyncio.run(get_date())
print(f"Current date: {date}")
```

Voir également *le même exemple*, écrit en utilisant des API de bas niveau.

19.1.6 Files d'attente (*queues*)

Code source : Lib/asyncore.py

Les files d'attente *asyncio* sont conçues pour être similaires aux classes du module `queue`. Bien que les files d'attente *asyncio* ne soient pas compatibles avec les programmes à multiples fils d'exécution, elles sont conçues pour être utilisées spécifiquement dans le code *async/await*.

Notez que les méthodes des files d'attente *asyncio* n'ont pas de paramètre *timeout* ; utilisez la fonction `asyncio.wait_for()` pour effectuer des opérations de file d'attente avec un délai d'attente.

Voir également la section *Exemples* ci-dessous.

File d'attente (*queue*)

class asyncio.**Queue**(*maxsize=0*)

File d'attente premier entré, premier sorti (FIFO pour *fist in, first out*).

Si *maxsize* est inférieur ou égal à zéro, la taille de la file d'attente est infinie. Si c'est un entier supérieur à `0`, alors `await put()` se bloque lorsque la file d'attente atteint *maxsize* jusqu'à ce qu'un élément soit supprimé par `get()`.

Contrairement à la bibliothèque standard multi-fils `queue`, la taille de la file d'attente est toujours connue et peut être renvoyée en appelant la méthode `qsize()`.

Modifié dans la version 3.10 : suppression du paramètre *loop*.

Cette classe n'est *pas compatible avec les fils d'exécution multiples*.

maxsize

Nombre d'éléments autorisés dans la file d'attente.

empty()

Renvoie `True` si la file d'attente est vide, `False` sinon.

full()

Renvoie `True` s'il y a `maxsize` éléments dans la file d'attente.

If the queue was initialized with `maxsize=0` (the default), then `full()` never returns `True`.

coroutine get()

Supprime et renvoie un élément de la file d'attente. Si la file d'attente est vide, attend qu'un élément soit disponible.

Raises `QueueShutDown` if the queue has been shut down and is empty, or if the queue has been shut down immediately.

get_nowait()

Renvoie un élément s'il y en a un immédiatement disponible, sinon lève `QueueEmpty`.

coroutine join()

Bloque jusqu'à ce que tous les éléments de la file d'attente aient été récupérés et traités.

Le nombre de tâches inachevées augmente chaque fois qu'un élément est ajouté à la file. Ce nombre diminue chaque fois qu'un fil d'exécution consommateur appelle `task_done()` pour indiquer que l'élément a été extrait et que tout le travail à effectuer dessus est terminé. Lorsque le nombre de tâches non terminées devient nul, `join()` débloque.

coroutine put(*item*)

Met un élément dans la file d'attente. Si la file d'attente est pleine, attend qu'un emplacement libre soit disponible avant d'ajouter l'élément.

Raises `QueueShutDown` if the queue has been shut down.

put_nowait(*item*)

Ajoute un élément dans la file d'attente sans bloquer.

Si aucun emplacement libre n'est immédiatement disponible, lève `QueueFull`.

qsize()

Renvoie le nombre d'éléments dans la file d'attente.

shutdown(*immediate=False*)

Shut down the queue, making `get()` and `put()` raise `QueueShutDown`.

By default, `get()` on a shut down queue will only raise once the queue is empty. Set *immediate* to true to make `get()` raise immediately instead.

All blocked callers of `put()` and `get()` will be unblocked. If *immediate* is true, a task will be marked as done for each remaining item in the queue, which may unblock callers of `join()`.

Ajouté dans la version 3.13.

task_done()

Indicate that a formerly enqueued work item is complete.

Used by queue consumers. For each `get()` used to fetch a work item, a subsequent call to `task_done()` tells the queue that the processing on the work item is complete.

Si un `join()` est actuellement bloquant, on reprendra lorsque tous les éléments auront été traités (ce qui signifie qu'un appel à `task_done()` a été effectué pour chaque élément qui a été `put()` dans la file).

`shutdown(immediate=True)` calls `task_done()` for each remaining item in the queue.

Lève une exception `ValueError` si elle est appelée plus de fois qu'il n'y avait d'éléments dans la file.

File avec priorité

class asyncio.**PriorityQueue**

Une variante de `Queue`; récupère les entrées par ordre de priorité (la plus basse en premier).

Les entrées sont généralement des *n*-uplets de la forme `(priority_number, data)`.

Pile (LIFO)

class asyncio.**LifoQueue**

Une variante de `Queue` qui récupère en premier les entrées les plus récemment ajoutées (dernier entré, premier sorti).

Exceptions

exception asyncio.**QueueEmpty**

Cette exception est levée lorsque la méthode *get_nowait()* est appelée sur une file d'attente vide.

exception asyncio.**QueueFull**

Exception levée lorsque la méthode *put_nowait()* est appelée sur une file d'attente qui a atteint sa *maxsize*.

exception asyncio.**QueueShutDown**

Exception raised when *put()* or *get()* is called on a queue which has been shut down.

Ajouté dans la version 3.13.

Exemples

Les files d'attente peuvent être utilisées pour répartir la charge de travail entre plusieurs tâches simultanées :

```
import asyncio
import random
import time


async def worker(name, queue):
    while True:
        # Get a "work item" out of the queue.
        sleep_for = await queue.get()

        # Sleep for the "sleep_for" seconds.
        await asyncio.sleep(sleep_for)

        # Notify the queue that the "work item" has been processed.
        queue.task_done()

        print(f'{name} has slept for {sleep_for:.2f} seconds')


async def main():
    # Create a queue that we will use to store our "workload".
    queue = asyncio.Queue()

    # Generate random timings and put them into the queue.
    total_sleep_time = 0
    for _ in range(20):
        sleep_for = random.uniform(0.05, 1.0)
        total_sleep_time += sleep_for
        queue.put_nowait(sleep_for)

    # Create three worker tasks to process the queue concurrently.
    tasks = []
    for i in range(3):
        task = asyncio.create_task(worker(f'worker-{i}', queue))
        tasks.append(task)

    # Wait until the queue is fully processed.
    started_at = time.monotonic()
    await queue.join()
    total_slept_for = time.monotonic() - started_at

    # Cancel our worker tasks.
```

(suite sur la page suivante)

(suite de la page précédente)

```
    for task in tasks:
        task.cancel()
    # Wait until all worker tasks are cancelled.
    await asyncio.gather(*tasks, return_exceptions=True)

    print('====')
    print(f'3 workers slept in parallel for {total_slept_for:.2f} seconds')
    print(f'total expected sleep time: {total_sleep_time:.2f} seconds')


asyncio.run(main())
```

19.1.7 Exceptions

Code source : Lib/asyncio/exceptions.py

---

exception asyncio.**TimeoutError**

Alias obsolète de `TimeoutError`, levée lorsque l'opération a dépassé le délai donné.

Modifié dans la version 3.11 : cette classe est devenue un alias de `TimeoutError`.

exception asyncio.**CancelledError**

L'opération a été annulée.

Cette exception peut être interceptée pour effectuer des opérations personnalisées lorsque des tâches asynchrones sont annulées. Dans presque toutes les situations, l'exception doit être relancée.

Modifié dans la version 3.8 : `CancelledError` is now a subclass of `BaseException` rather than `Exception`.

exception asyncio.**InvalidStateError**

État interne non valide de `Task` ou `Future`.

Peut être levée dans des situations telles que la définition d'une valeur de résultat pour un objet *Future* qui a déjà une valeur de résultat définie.

exception asyncio.**SendfileNotAvailableError**

L'appel système *sendfile* n'est pas disponible pour le connecteur ou le type de fichier donné.

Sous-classe de `RuntimeError`.

exception asyncio.**IncompleteReadError**

L'opération de lecture demandée s'est terminée avant d'être complète.

Levée par les API de *flux asyncio*.

Cette exception est une sous-classe de `EOFError`.

expected

Nombre total (`int`) d'octets attendus.

partial

Chaîne de `bytes` lue avant que la fin du flux ne soit atteinte.

exception asyncio.**LimitOverrunError**

La limite de taille de tampon a été atteinte lors de la recherche d'un séparateur.

Levée par les API *flux asyncio*.

consumed

Nombre total d'octets à consommer.

19.1.8 Call Graph Introspection

Source code : Lib/asyncio/graph.py

---

asyncio has powerful runtime call graph introspection utilities to trace the entire call graph of a running *coroutine* or *task*, or a suspended *future*. These utilities and the underlying machinery can be used from within a Python program or by external profilers and debuggers.

Ajouté dans la version 3.14.

`asyncio.print_call_graph`(*future=None*, /, *, *file=None*, *depth=1*, *limit=None*)

Print the async call graph for the current task or the provided `Task` or `Future`.

This function prints entries starting from the top frame and going down towards the invocation point.

The function receives an optional *future* argument. If not passed, the current running task will be used.

If the function is called on *the current task*, the optional keyword-only *depth* argument can be used to skip the specified number of frames from top of the stack.

If the optional keyword-only *limit* argument is provided, each call stack in the resulting graph is truncated to include at most `abs(limit)` entries. If *limit* is positive, the entries left are the closest to the invocation point. If *limit* is negative, the topmost entries are left. If *limit* is omitted or `None`, all entries are present. If *limit* is `0`, the call stack is not printed at all, only "awaited by" information is printed.

If *file* is omitted or `None`, the function will print to `sys.stdout`.

Example :

The following Python code :

```
import asyncio

async def test():
    asyncio.print_call_graph()

async def main():
    async with asyncio.TaskGroup() as g:
        g.create_task(test(), name='test')

asyncio.run(main())
```

will print :

```
* Task(name='test', id=0x1039f0fe0)
+ Call stack:
|   File 't2.py', line 4, in async test()
+ Awaited by:
   * Task(name='Task-1', id=0x103a5e060)
      + Call stack:
      |   File 'taskgroups.py', line 107, in async TaskGroup.__aexit__()
      |   File 't2.py', line 7, in async main()
```

`asyncio.format_call_graph`(*future=None*, /, *, *depth=1*, *limit=None*)

Like `print_call_graph()`, but returns a string. If *future* is `None` and there's no current task, the function returns an empty string.

`asyncio.capture_call_graph`(*future=None*, /, *, *depth=1*, *limit=None*)

Capture the async call graph for the current task or the provided `Task` or `Future`.

The function receives an optional *future* argument. If not passed, the current running task will be used. If there's no current task, the function returns `None`.

If the function is called on *the current task*, the optional keyword-only *depth* argument can be used to skip the specified number of frames from top of the stack.

Returns a `FutureCallGraph` data class object :

— `FutureCallGraph(future, call_stack, awaited_by)`

Where *future* is a reference to a `Future` or a `Task` (or their subclasses.)

`call_stack` is a tuple of `FrameCallGraphEntry` objects.

`awaited_by` is a tuple of `FutureCallGraph` objects.

— `FrameCallGraphEntry(frame)`

Where *frame* is a frame object of a regular Python function in the call stack.

Low level utility functions

To introspect an async call graph asyncio requires cooperation from control flow structures, such as `shield()` or `TaskGroup`. Any time an intermediate `Future` object with low-level APIs like `Future.add_done_callback()` is involved, the following two functions should be used to inform asyncio about how exactly such intermediate future objects are connected with the tasks they wrap or control.

`asyncio.`**`future_add_to_awaited_by`**(*future*, *waiter*, /)

Record that *future* is awaited on by *waiter*.

Both *future* and *waiter* must be instances of `Future` or `Task` or their subclasses, otherwise the call would have no effect.

A call to `future_add_to_awaited_by()` must be followed by an eventual call to the `future_discard_from_awaited_by()` function with the same arguments.

`asyncio.`**`future_discard_from_awaited_by`**(*future*, *waiter*, /)

Record that *future* is no longer awaited on by *waiter*.

Both *future* and *waiter* must be instances of `Future` or `Task` or their subclasses, otherwise the call would have no effect.

19.1.9 Boucle d'évènements

Code source : Lib/asyncio/events.py, Lib/asyncio/base_events.py

---

Préface

La boucle d'événements est au cœur de chaque application *asyncio*. Les boucles d'événements exécutent des tâches et des rappels asynchrones, effectuent des opérations d'entrée-sorite réseau et exécutent des sous-processus.

Les développeurs d'applications doivent généralement utiliser les fonctions *asyncio* de haut niveau, telles que `asyncio.run()`, et ne doivent que rarement référencer l'objet boucle ou appeler ses méthodes. Cette section est principalement destinée aux auteurs de code, de bibliothèques et de cadriciels de bas niveau, qui ont besoin d'un contrôle plus précis sur le comportement de la boucle d'événements.

Obtention d'une boucle d'évènements

Les fonctions de bas niveau suivantes peuvent être utilisées pour obtenir, définir ou créer une boucle d'événements :

`asyncio.`**`get_running_loop`**()

Renvoie la boucle d'événements en cours d'exécution dans le fil actuel du système d'exploitation.

Lève une `RuntimeError` s'il n'y a pas de boucle d'événements en cours d'exécution.

Cette fonction ne peut être appelée qu'à partir d'une coroutine ou d'une fonction de rappel.

Ajouté dans la version 3.7.

`asyncio.get_event_loop()`

Arrête l'exécution de la boucle d'évènements.

Lorsqu'elle est appelée depuis une coroutine ou une fonction de rappel (par exemple planifiée avec *call_soon* ou une API similaire), cette fonction renvoie toujours la boucle d'événement en cours.

S'il n'y a pas de boucle d'événement en cours d'exécution, la fonction renvoie le résultat de l'appel `get_event_loop_policy().get_event_loop()`.

Étant donné que cette fonction a un comportement plutôt complexe (en particulier lorsque des politiques de boucle d'événements personnalisées sont utilisées), l'utilisation de la fonction `get_running_loop()` est préférable à `get_event_loop()` dans les coroutines et les fonctions de rappel.

Comme indiqué ci-dessus, envisagez d'utiliser la fonction de haut niveau `asyncio.run()`, au lieu d'utiliser ces fonctions de bas niveau pour créer et fermer manuellement une boucle d'événements.

Modifié dans la version 3.14 : Raises a `RuntimeError` if there is no current event loop.

> **Note**
>
> The `asyncio` policy system is deprecated and will be removed in Python 3.16 ; from there on, this function will always return the running event loop.

`asyncio.set_event_loop(loop)`

Définit *loop* comme boucle d'événements actuelle pour le fil d'exécution actuel du système d'exploitation.

Obsolète depuis la version 3.14 : The `set_event_loop()` function is deprecated and will be removed in Python 3.16.

`asyncio.new_event_loop()`

Crée et renvoie un nouvel objet de boucle d'événements.

Notez que le comportement des fonctions `get_event_loop()`, `set_event_loop()` et `new_event_loop()` peut être modifié en *définissant une politique de boucle d'événement personnalisée*.

Sommaire

Cette page de documentation contient les sections suivantes :

- la section *Event Loop Methods* est la documentation de référence des API de boucle d'événements ;
- la section *Callback Handles* documente les instances `Handle` et `TimerHandle` qui sont renvoyées par les méthodes de planification telles que `loop.call_soon()` et `loop.call_later()` ;
- la section *Server Objects* documente les types renvoyés par les méthodes de boucle d'événements comme `loop.create_server()` ;
- la section *Event Loop Implementations* documente les classes `SelectorEventLoop` et `ProactorEventLoop` ;
- la section *Exemples* montre comment travailler avec certaines API de boucle d'événements.

Méthodes de la boucle d'évènements

Les boucles d'événements ont des API **de bas niveau** pour les éléments suivants :

- *Démarrer et arrêter une boucle d'évènements*
- *Planification des fonctions de rappel*
- *Planification des rappels différés*
- *Création de* Futures *et des tâches*
- *Création de connexions*
- *Création de serveurs*
- *Transfert de fichiers*
- *Passage du flux en TLS*

— *Surveillance de descripteur de fichier*
— *Travail direct avec des objets* socket
— *DNS*
— *Travail avec des tubes (*pipes*)*
— *Signaux Unix*
— *Exécution de code dans des pools de threads ou de processus*
— *API de gestion d'erreur*
— *Activation du mode débogage*
— *Exécution de sous-processus*

Démarrer et arrêter une boucle d'évènements

`loop.run_until_complete(future)`

Lance la boucle jusqu'à ce que *future* (une instance de `Future`) soit terminée.

Si l'argument est un objet *coroutine*, il est implicitement programmé pour s'exécuter en tant que `asyncio.Task`.

Renvoie le résultat du *Future* ou lève son exception.

`loop.run_forever()`

Exécute la boucle d'événement jusqu'à ce que `stop()` soit appelée.

If `stop()` is called before `run_forever()` is called, the loop will poll the I/O selector once with a timeout of zero, run all callbacks scheduled in response to I/O events (and those that were already scheduled), and then exit.

Si `stop()` est appelée pendant que `run_forever()` est en cours d'exécution, la boucle exécute le lot actuel de rappels puis se termine. Notez que les nouveaux rappels programmés par fonctions de rappel ne s'exécuteront pas dans ce cas ; à la place, ils s'exécuteront la prochaine fois que `run_forever()` ou `run_until_complete()` sera appelée.

`loop.stop()`

Arrête l'exécution de la boucle d'évènements.

`loop.is_running()`

Renvoie `True` si la boucle d'évènements est démarrée.

`loop.is_closed()`

Renvoie `True` si la boucle d'évènements est arrêtée.

`loop.close()`

Arrête la boucle d'évènements.

La boucle ne doit pas être en cours d'exécution lorsque cette fonction est appelée. Tous les rappels en attente seront ignorés.

Cette méthode efface toutes les files d'attente et arrête l'exécuteur, mais n'attend pas que l'exécuteur se termine.

Cette méthode est idempotente et irréversible. Aucune autre méthode ne doit être appelée après la fermeture de la boucle d'événements.

coroutine `loop.shutdown_asyncgens()`

Schedule all currently open *asynchronous generator* objects to close with an `aclose()` call. After calling this method, the event loop will issue a warning if a new asynchronous generator is iterated. This should be used to reliably finalize all scheduled asynchronous generators.

Notez qu'il n'est pas nécessaire d'appeler cette fonction lorsque `asyncio.run()` est utilisée.

Exemple :

```
try:
    loop.run_forever()
finally:
    loop.run_until_complete(loop.shutdown_asyncgens())
    loop.close()
```

Ajouté dans la version 3.6.

coroutine loop.**shutdown_default_executor**(*timeout=None*)

Planifie la fermeture de l'exécuteur par défaut et attend que tous les fils se rejoignent dans le `ThreadPoolExecutor`. Une fois cette méthode appelée, l'utilisation de l'exécuteur par défaut avec `loop.run_in_executor()` lève une `RuntimeError`.

The *timeout* parameter specifies the amount of time (in `float` seconds) the executor will be given to finish joining. With the default, `None`, the executor is allowed an unlimited amount of time.

If the *timeout* is reached, a `RuntimeWarning` is emitted and the default executor is terminated without waiting for its threads to finish joining.

> **Note**
>
> n'appelez pas cette méthode lorsque vous utilisez `asyncio.run()`, car cette dernière gère automatiquement l'arrêt de l'exécuteur par défaut.

Ajouté dans la version 3.9.

Modifié dans la version 3.12 : Added the *timeout* parameter.

Planification des fonctions de rappel

loop.**call_soon**(*callback*, **args*, *context=None*)

Définit la *fonction de rappel* *callback* à appeler avec les arguments *args* à la prochaine itération de la boucle d'événements.

Renvoie une instance de `asyncio.Handle`, qui pourra être utilisée ultérieurement pour annuler le rappel.

Les fonctions de rappels sont appelées dans l'ordre dans lequel elles sont enregistrées. Chaque fonction de rappel sera appelée exactement une fois.

L'argument facultatif nommé uniquement *context* spécifie un `contextvars.Context` personnalisé pour le *callback* à exécuter. Les rappels utilisent le contexte actuel lorsqu'aucun *context* n'est fourni.

Contrairement à `call_soon_threadsafe()`, cette méthode n'est pas compatible avec les programmes à fils d'exécution multiples.

loop.**call_soon_threadsafe**(*callback*, **args*, *context=None*)

Une variante compatible avec les programmes à fils d'exécution multiples de `call_soon()`. Lors de la planification de rappels à partir d'un autre fil d'exécution, cette fonction *doit* être utilisée, puisque `call_soon()` n'est pas thread-safe.

This function is safe to be called from a reentrant context or signal handler, however, it is not safe or fruitful to use the returned handle in such contexts.

Lève `RuntimeError` si elle est appelée sur une boucle qui a été fermée. Cela peut se produire sur un fil secondaire lorsque l'application principale se ferme.

Voir la section *exécution concurrente et multi-fils d'exécution* de la documentation.

Modifié dans la version 3.7 : le paramètre nommé uniquement *context* a été ajouté. Voir **PEP 567** pour plus de détails.

Note

la plupart des fonctions d'ordonnancement `asyncio` n'autorisent pas le passage d'arguments nommés. Pour le faire, utilisez `functools.partial()`

```
# will schedule "print("Hello", flush=True)"
loop.call_soon(
    functools.partial(print, "Hello", flush=True))
```

L'utilisation d'objets partiels est généralement plus pratique que l'utilisation de lambdas, car *asyncio* peut mieux rendre les objets partiels dans les messages de débogage et d'erreur.

Planification des rappels différés

La boucle d'événements fournit des mécanismes pour programmer les fonctions de rappel à appeler à un moment donné dans le futur. La boucle d'événements utilise des horloges monotones pour suivre le temps.

`loop.call_later`(*delay*, *callback*, **args*, *context=None*)

Planifie le rappel *callback* à appeler après *delay* secondes (peut être un entier ou un flottant).

Une instance de `asyncio.TimerHandle` est renvoyée et peut être utilisée pour annuler le rappel.

callback sera appelé exactement une fois. Si deux rappels sont programmés exactement à la même heure, l'ordre dans lequel ils sont appelés n'est pas défini.

L'argument positionnel facultatif *args* sera transmis au rappel lorsqu'il sera appelé. Si vous voulez que le rappel soit appelé avec des arguments nommés, utilisez `functools.partial()`.

Un argument facultatif *context* nommé uniquement permet de spécifier un `contextvars.Context` personnalisé pour le *callback* à exécuter. Le contexte actuel est utilisé lorsqu'aucun *context* n'est fourni.

Modifié dans la version 3.7 : le paramètre nommé uniquement *context* a été ajouté. Voir **PEP 567** pour plus de détails.

Modifié dans la version 3.8 : dans Python 3.7 et versions antérieures avec l'implémentation de la boucle d'événements par défaut, le *delay* ne pouvait pas dépasser un jour. Cela a été corrigé dans Python 3.8.

`loop.call_at`(*when*, *callback*, **args*, *context=None*)

Planifie l'appel de *callback* à l'horodatage absolu donné *when* (un *int* ou un *float*), en utilisant la même référence de temps que `loop.time()`.

Le comportement de cette méthode est le même que `call_later()`.

Une instance de `asyncio.TimerHandle` est renvoyée et peut être utilisée pour annuler le rappel.

Modifié dans la version 3.7 : le paramètre nommé uniquement *context* a été ajouté. Voir **PEP 567** pour plus de détails.

Modifié dans la version 3.8 : dans Python 3.7 et versions antérieures avec l'implémentation de la boucle d'événements par défaut, la différence entre *when* et l'heure actuelle ne pouvait pas dépasser un jour. Cela a été corrigé dans Python 3.8.

`loop.time`()

Renvoie l'heure actuelle, sous la forme d'une valeur `float`, selon l'horloge monotone interne de la boucle d'événements.

Note

Modifié dans la version 3.8 : dans Python 3.7 et les versions antérieures, les délais d'expiration (relatif *delay* ou absolu *when*) ne doivent pas dépasser un jour. Cela a été corrigé dans Python 3.8.

Voir aussi

la fonction `asyncio.sleep()`.

Création de *Futures* et des tâches

`loop.create_future()`

Crée un objet `asyncio.Future` attaché à la boucle d'événements.

C'est la méthode préférée pour créer des *Futures* avec *asyncio*. Cela permet aux boucles d'événements tierces de fournir des implémentations alternatives de l'objet *Future* (avec de meilleures performances ou instrumentation).

Ajouté dans la version 3.5.2.

`loop.create_task(coro, *, name=None, context=None)`

Planifie l'exécution de *coroutine* *coro*. Renvoie un objet `Task`.

Les boucles d'événements tierces peuvent utiliser leur propre sous-classe de `Task` pour l'interopérabilité. Dans ce cas, le type de résultat est une sous-classe de `Task`.

Si l'argument *name* est fourni et non `None`, il est défini comme le nom de la tâche en utilisant `Task.set_name()`.

Argument facultatif *context* nommé uniquement qui permet de spécifier un `contextvars.Context` personnalisé pour la *coro* à exécuter. La copie de contexte actuel est créée lorsqu'aucun *context* n'est fourni.

Modifié dans la version 3.8 : ajout du paramètre `name`.

Modifié dans la version 3.11 : ajout du paramètre `context`.

`loop.set_task_factory(factory)`

Définit une fabrique de tâches qui sera utilisée par `loop.create_task()`.

If *factory* is `None` the default task factory will be set. Otherwise, *factory* must be a *callable* with the signature matching `(loop, coro, **kwargs)`, where *loop* is a reference to the active event loop, and *coro* is a coroutine object. The callable must pass on all *kwargs*, and return a `asyncio.Task`-compatible object.

`loop.get_task_factory()`

Renvoie une fabrique de tâches ou `None` si celle par défaut est utilisée.

Création de connexions

coroutine `loop.create_connection(protocol_factory, host=None, port=None, *, ssl=None, family=0, proto=0, flags=0, sock=None, local_addr=None, server_hostname=None, ssl_handshake_timeout=None, ssl_shutdown_timeout=None, happy_eyeballs_delay=None, interleave=None, all_errors=False)`

Ouvre un flux de transport connecté à l'adresse spécifiée par *host* et *port*.

The socket family can be either `AF_INET` or `AF_INET6` depending on *host* (or the *family* argument, if provided).

The socket type will be `SOCK_STREAM`.

protocol_factory doit être un appelable renvoyant un protocole gérant le *protocole asyncio*.

Cette méthode tente d'établir la connexion en arrière-plan. En cas de succès, elle renvoie une paire `(transport, protocol)`.

Le synopsis chronologique de l'opération sous-jacente est le suivant :

1. La connexion est établie et un *transport asyncio* est créé pour cela.
2. *protocol_factory* est appelée sans arguments et doit renvoyer une instance de *protocol asyncio*.

3. L'instance de protocole est couplée au transport en appelant sa méthode `connection_made()`.
4. Un *n*-uplet `(transport, protocol)` est renvoyé en cas de succès.

Le transport créé est un flux bidirectionnel dépendant de l'implémentation.

Autres arguments :

— *ssl* : s'il est donné et non faux, un transport SSL/TLS est créé (par défaut un transport TCP simple est créé). Si *ssl* est un objet `ssl.SSLContext`, ce contexte est utilisé pour créer le transport ; si *ssl* est `True`, un contexte par défaut renvoyé par `ssl.create_default_context()` est utilisé.

> **Voir aussi**
>
> *Considérations sur la sécurité SSL/TLS*

— *server_hostname* définit ou remplace le nom d'hôte auquel le certificat du serveur cible sera comparé. Ne doit être passé que si *ssl* n'est pas `None`. Par défaut, la valeur de l'argument *host* est utilisée. Si *host* est vide, il n'y a pas de valeur par défaut et vous devez transmettre une valeur pour *server_hostname*. Si *server_hostname* est une chaîne vide, la correspondance du nom d'hôte est désactivée (ce qui constitue un risque de sécurité sérieux, permettant des attaques potentielles de type « homme du milieu »).

— *family*, *proto*, *flags* sont facultatifs et sont la famille d'adresse, le protocole et les drapeaux à transmettre à *getaddrinfo()* pour la résolution de *host*. S'ils sont fournis, ils doivent tous être des entiers provenant des constantes du module `socket`.

— *happy_eyeballs_delay*, s'il est fourni, active Happy Eyeballs pour cette connexion. Il doit s'agir d'un nombre à virgule flottante représentant le temps d'attente en secondes pour qu'une tentative de connexion se termine, avant de démarrer la prochaine tentative en parallèle. Il s'agit du « délai de tentative de connexion » tel que défini dans la **RFC 8305**. Une valeur par défaut raisonnable recommandée par la RFC est `0.25` (250 millisecondes).

— *interleave* contrôle la réorganisation des adresses lorsqu'un nom d'hôte se résout en plusieurs adresses IP. S'il vaut `0` ou n'est pas spécifié, aucune réorganisation n'est effectuée et les adresses sont essayées dans l'ordre renvoyé par `getaddrinfo()`. Si un entier positif est spécifié, les adresses sont entrelacées par famille d'adresses et l'entier donné est interprété comme "First Address Family Count" tel que défini dans la **RFC 8305**. La valeur par défaut est `0` si *happy_eyeballs_delay* n'est pas spécifié, et `1` si c'est le cas.

— *sock*, s'il est fourni, doit être un objet `socket.socket` existant et déjà connecté à utiliser par le transport. Si *sock* est donné, aucun des *host*, *port*, *family*, *proto*, *flags*, *happy_eyeballs_delay*, *interleave* et *local_addr* ne doit être spécifié.

> **Note**
>
> l'argument *sock* transfère la propriété du connecteur au transport créé. Pour fermer le connecteur, appelez la méthode `close()` du transport.

— *local_addr*, s'il est fourni, est un *n*-uplet `(local_host, local_port)` utilisé pour lier le connecteur localement. *local_host* et *local_port* sont recherchés en utilisant `getaddrinfo()`, de la même manière que *host* et *port*.

— *ssl_handshake_timeout* est (pour une connexion TLS) le temps en secondes à attendre que la poignée de main TLS se termine avant d'abandonner la connexion. `60.0` secondes si `None` (par défaut).

— *ssl_shutdown_timeout* est le temps en secondes à attendre que l'arrêt SSL se termine avant d'abandonner la connexion. `30.0` secondes si `None` (par défaut).

— *all_errors* determines what exceptions are raised when a connection cannot be created. By default, only a single `Exception` is raised : the first exception if there is only one or all errors have same message, or a single `OSError` with the error messages combined. When `all_errors` is `True`, an `ExceptionGroup` will be raised containing all exceptions (even if there is only one).

Modifié dans la version 3.5 : ajout de la prise en charge de SSL/TLS dans `ProactorEventLoop`.

Modifié dans la version 3.6 : The socket option *socket.TCP_NODELAY* is set by default for all TCP connections.

Modifié dans la version 3.7 : ajout du paramètre *ssl handshake timeout*

Modifié dans la version 3.8 : ajout des paramètres *happy_eyeballs_delay* et *interleave*.

Algorithme Happy Eyeballs : « succès avec les hôtes à double pile ». Lorsque le chemin et le protocole IPv4 d'un

serveur fonctionnent, mais que le chemin et le protocole IPv6 du serveur ne fonctionnent pas, une application cliente à double pile subit un retard de connexion important par rapport à un client IPv4 uniquement. Ceci n'est pas souhaitable car cela entraîne une moins bonne expérience utilisateur pour le client à double pile. Ce document spécifie les exigences pour les algorithmes qui réduisent ce délai visible par l'utilisateur et fournit un algorithme correspondant.

For more information : https://datatracker.ietf.org/doc/html/rfc6555

Modifié dans la version 3.11 : ajout du paramètre *ssl shutdown timeout*

Modifié dans la version 3.12 : *all_errors* was added.

Voir aussi

la fonction `open_connection()` est une API alternative de haut niveau. Elle renvoie une paire de (`StreamReader`, `StreamWriter`) qui peut être utilisée directement dans le code *async/wait*.

coroutine `loop.`**`create_datagram_endpoint`**(*protocol_factory*, *local_addr=None*, *remote_addr=None*, *, *family=0*, *proto=0*, *flags=0*, *reuse_port=None*, *allow_broadcast=None*, *sock=None*)

Création d'une connexion par datagramme

The socket family can be either `AF_INET`, `AF_INET6`, or `AF_UNIX`, depending on *host* (or the *family* argument, if provided).

The socket type will be `SOCK_DGRAM`.

protocol_factory doit être un appelable gérant le *protocole asyncio*.

Un *n*-uplet `(transport, protocol)` est renvoyé en cas de succès.

Autres arguments :

— *local_addr*, s'il est fourni, est un *n*-uplet `(local_host, local_port)` utilisé pour lier le connecteur localement. Le *local_host* et le *local_port* sont recherchés en utilisant `getaddrinfo()`.

— *remote_addr*, s'il est fourni, est un *n*-uplet `(remote_host, remote_port)` utilisé pour se connecter à une adresse distante. Le *remote_host* et le *remote_port* sont recherchés en utilisant `getaddrinfo()`.

— *family*, *proto*, *flags* sont facultatifs et représentent la famille d'adresse, le protocole et les drapeaux à transmettre à `getaddrinfo()` pour la résolution *host*. S'ils sont fournis, ils doivent tous être des entiers provenant des constantes du module `socket`.

— *reuse_port* tells the kernel to allow this endpoint to be bound to the same port as other existing endpoints are bound to, so long as they all set this flag when being created. This option is not supported on Windows and some Unixes. If the *socket.SO_REUSEPORT* constant is not defined then this capability is unsupported.

— *allow_broadcast* indique au noyau d'autoriser ce point de terminaison à envoyer des messages à l'adresse de *broadcast*.

— *sock* peut éventuellement être spécifié afin d'utiliser un objet `socket.socket` préexistant, déjà connecté, à utiliser par le transport. Si spécifié, *local_addr* et *remote_addr* doivent être omis (doit être `None`).

Note

l'argument *sock* transfère la propriété du connecteur au transport créé. Pour fermer le connecteur, appelez la méthode `close()` du transport.

Voir les exemples *Client écho en UDP* et *Serveur écho en UDP*.

Modifié dans la version 3.4.4 : les paramètres *family*, *proto*, *flags*, *reuse_address*, *reuse_port*, *allow_broadcast* et *sock* ont été ajoutés.

Modifié dans la version 3.8 : prise en charge sur Windows.

Modifié dans la version 3.8.1 : The *reuse_address* parameter is no longer supported, as using *socket.SO_REUSEADDR* poses a significant security concern for UDP. Explicitly passing `reuse_address=True` will raise an exception.

Lorsque plusieurs processus avec des UID différents attribuent des connecteurs à une adresse de connecteur UDP identique avec `SO_REUSEADDR`, les paquets entrants peuvent être distribués de manière aléatoire entre les connecteurs.

For supported platforms, *reuse_port* can be used as a replacement for similar functionality. With *reuse_port*, *socket.SO_REUSEPORT* is used instead, which specifically prevents processes with differing UIDs from assigning sockets to the same socket address.

Modifié dans la version 3.11 : The *reuse_address* parameter, disabled since Python 3.8.1, 3.7.6 and 3.6.10, has been entirely removed.

coroutine `loop.`**`create_unix_connection`**(*protocol_factory*, *path=None*, *, *ssl=None*, *sock=None*, *server_hostname=None*, *ssl_handshake_timeout=None*, *ssl_shutdown_timeout=None*)

Crée une connexion Unix

The socket family will be `AF_UNIX`; socket type will be `SOCK_STREAM`.

Un *n*-uplet `(transport, protocol)` est renvoyé en cas de succès.

path est le nom d'un connecteur de domaine Unix et est obligatoire, sauf si un paramètre *sock* est spécifié. Les connecteurs Unix abstraits, les chemins `str`, `bytes` et `Path` sont pris en charge.

Voir la documentation de la méthode `loop.create_connection()` pour plus d'informations sur les arguments de cette méthode.

Availability : Unix.

Modifié dans la version 3.7 : ajout du paramètre *ssl_handshake_timeout*. Le paramètre *chemin* peut désormais être un *objet simili-chemin*.

Modifié dans la version 3.11 : ajout du paramètre *ssl shutdown timeout*

Création de serveurs

coroutine `loop.`**`create_server`**(*protocol_factory*, *host=None*, *port=None*, *, *family=socket.AF_UNSPEC*, *flags=socket.AI_PASSIVE*, *sock=None*, *backlog=100*, *ssl=None*, *reuse_address=None*, *reuse_port=None*, *keep_alive=None*, *ssl_handshake_timeout=None*, *ssl_shutdown_timeout=None*, *start_serving=True*)

Create a TCP server (socket type `SOCK_STREAM`) listening on *port* of the *host* address.

Renvoie un objet `Server`.

Arguments :

- *protocol_factory* doit être un appelable gérant le *protocole asyncio*.
- Le paramètre *host* peut être défini sur plusieurs types qui déterminent où le serveur écoute :
  - Si *host* est une chaîne, le serveur TCP est lié à une seule interface réseau spécifiée par *host*.
  - Si *host* est une séquence de chaînes, le serveur TCP est lié à toutes les interfaces réseau spécifiées par la séquence.
  - Si *host* est une chaîne vide ou `None`, toutes les interfaces sont prises en compte et une liste de plusieurs connecteurs est renvoyée (probablement une pour IPv4 et une autre pour IPv6).
- Le paramètre *port* peut être défini pour spécifier sur quel port le serveur doit écouter. Si `0` ou `None` (la valeur par défaut), un port inutilisé aléatoire est sélectionné (notez que si *host* se résout en plusieurs interfaces réseau, un port aléatoire différent est sélectionné pour chaque interface).
- *family* can be set to either `socket.AF_INET` or `AF_INET6` to force the socket to use IPv4 or IPv6. If not set, the *family* will be determined from host name (defaults to `AF_UNSPEC`).
- *flags* est un masque de bits pour `getaddrinfo()`.
- *sock* peut éventuellement être spécifié afin d'utiliser un objet connecteur préexistant. S'il est spécifié, *host* et *port* ne doivent pas être spécifiés.

> **Note**
>
> l'argument *sock* transfère la propriété du connecteur au serveur créé. Pour fermer le connecteur, appelez la méthode `close()` du serveur.

— *backlog* est le nombre maximum de connexions en file d'attente passées à `listen()` (par défaut à 100).
— *ssl* peut être défini sur une instance `SSLContext` pour activer TLS sur les connexions acceptées.
— *reuse_address* indique au noyau de réutiliser un connecteur local dans l'état `TIME_WAIT`, sans attendre l'expiration de son délai d'attente naturel. S'il n'est pas spécifié, il est automatiquement défini sur `True` sous Unix.
— *reuse_port* indique au noyau d'autoriser ce point de terminaison à être lié au même port que les autres points de terminaison existants, tant qu'ils définissent tous cet indicateur lors de leur création. Cette option n'est pas prise en charge sous Windows.
— *keep_alive* set to `True` keeps connections active by enabling the periodic transmission of messages.

Modifié dans la version 3.13 : Added the *keep_alive* parameter.

— *ssl_handshake_timeout* est (pour un serveur TLS) le temps en secondes à attendre que la poignée de main TLS se termine avant d'abandonner la connexion. `60.0` secondes si `None` (par défaut).
— *ssl_shutdown_timeout* est le temps en secondes à attendre que l'arrêt SSL se termine avant d'abandonner la connexion. `30.0` secondes si `None` (par défaut).
— *start_serving* défini à `True` (valeur par défaut) fait que le serveur créé commence immédiatement à accepter les connexions. Lorsqu'il est défini sur `False`, l'utilisateur doit attendre sur `Server.start_serving()` ou `Server.serve_forever()` pour que le serveur commence à accepter les connexions.

Modifié dans la version 3.5 : ajout de la prise en charge de SSL/TLS dans `ProactorEventLoop`.

Modifié dans la version 3.5.1 : le paramètre *host* peut être une séquence de chaînes.

Modifié dans la version 3.6 : Added *ssl_handshake_timeout* and *start_serving* parameters. The socket option *socket.TCP_NODELAY* is set by default for all TCP connections.

Modifié dans la version 3.11 : ajout du paramètre *ssl shutdown timeout*

> **Voir aussi**
>
> la fonction `start_server()` est une API alternative de niveau supérieur qui renvoie une paire de `StreamReader` et `StreamWriter` qui peut être utilisée dans un code *async/wait*.

coroutine loop.**create_unix_server**(*protocol_factory*, *path=None*, *, *sock=None*, *backlog=100*, *ssl=None*, *ssl_handshake_timeout=None*, *ssl_shutdown_timeout=None*, *start_serving=True*, *cleanup_socket=True*)

Similar to `loop.create_server()` but works with the `AF_UNIX` socket family.

path est le nom d'un connecteur de domaine Unix et est obligatoire, sauf si un argument *sock* est fourni. Les connecteurs Unix abstraits, les chemins `str`, `bytes` et `Path` sont pris en charge.

If *cleanup_socket* is true then the Unix socket will automatically be removed from the filesystem when the server is closed, unless the socket has been replaced after the server has been created.

Voir la documentation de la méthode `loop.create_server()` pour plus d'informations sur les arguments de cette méthode.

Availability : Unix.

Modifié dans la version 3.7 : ajout des paramètres *ssl_handshake_timeout* et *start_serving*. Le paramètre *path* peut maintenant être un objet `Path`.

Modifié dans la version 3.11 : ajout du paramètre *ssl shutdown timeout*

Modifié dans la version 3.13 : Added the *cleanup_socket* parameter.

coroutine loop.**connect_accepted_socket**(*protocol_factory*, *sock*, *, *ssl=None*, *ssl_handshake_timeout=None*, *ssl_shutdown_timeout=None*)

Enveloppe une connexion déjà acceptée dans une paire transport/protocole.

Cette méthode peut être utilisée par les serveurs qui acceptent les connexions en dehors d'*asyncio* mais qui utilisent *asyncio* pour les gérer.

Paramètres :
— *protocol_factory* doit être un appelable gérant le *protocole asyncio*.
— *sock* est un objet connecteur préexistant renvoyé par `socket.accept`.

> **Note**
>
> l'argument *sock* transfère la propriété du connecteur au transport créé. Pour fermer le connecteur, appelez la méthode `close()` du transport.

— *ssl* peut être défini sur une `SSLContext` pour activer SSL sur les connexions acceptées.
— *ssl_handshake_timeout* est (pour une connexion SSL) le temps en secondes à attendre que la poignée de main SSL se termine avant d'abandonner la connexion. `60.0` secondes si `None` (par défaut).
— *ssl_shutdown_timeout* est le temps en secondes à attendre que l'arrêt SSL se termine avant d'abandonner la connexion. `30.0` secondes si `None` (par défaut).

Renvoie une paire `(transport, protocole)`.

Ajouté dans la version 3.5.3.

Modifié dans la version 3.7 : ajout du paramètre *ssl handshake timeout*

Modifié dans la version 3.11 : ajout du paramètre *ssl shutdown timeout*

Transfert de fichiers

coroutine `loop.`**`sendfile`**(*transport*, *file*, *offset=0*, *count=None*, *, *fallback=True*)

Envoie *file* via *transport*. Renvoie le nombre total d'octets envoyés.

La méthode utilise `os.sendfile()` (hautes performances) si elle est disponible.

file doit être un objet fichier normal ouvert en mode binaire.

offset indique où commencer la lecture du fichier. Si spécifié, *count* est le nombre total d'octets à transmettre, par opposition à l'envoi du fichier jusqu'à ce que EOF soit atteint. La position du fichier est toujours mise à jour, même lorsque cette méthode génère une erreur. `file.tell()` peut être utilisée pour obtenir le nombre d'octets réellement envoyés.

fallback défini sur `True` permet à *asyncio* de lire et d'envoyer manuellement le fichier lorsque la plateforme ne prend pas en charge l'appel système *sendfile* (par exemple, Windows ou connecteur SSL sous Unix).

Lève `SendfileNotAvailableError` si le système ne prend pas en charge l'appel système *sendfile* et que *fallback* est `False`.

Ajouté dans la version 3.7.

Passage du flux en TLS

coroutine `loop.`**`start_tls`**(*transport*, *protocol*, *sslcontext*, *, *server_side=False*, *server_hostname=None*, *ssl_handshake_timeout=None*, *ssl_shutdown_timeout=None*)

Convertit une connexion existante en connexion TLS.

Crée une instance de codeur-décodeur TLS et l'insère entre le *transport* et le *protocol*. Le codeur-décodeur implémente à la fois le protocole vers le *transport* et le transport vers le *protocol*.

Renvoie l'instance à deux interfaces créée. Après *await*, le *protocol* doit cesser d'utiliser le *transport* d'origine et communiquer avec l'objet renvoyé uniquement parce que le codeur met en cache les données côté *protocol* et échange sporadiquement des paquets de session TLS supplémentaires avec *transport*.

In some situations (e.g. when the passed transport is already closing) this may return `None`.

Paramètres :

— *transport* et *protocol* que des méthodes comme `create_server()` et `create_connection()` renvoient.
— *sslcontext* : une instance configurée de `SSLContext`.
— *server_side* passe à `True` lorsqu'une connexion côté serveur est mise à jour (comme celle créée par `create_server()`).
— *server_hostname* : définit ou remplace le nom d'hôte auquel le certificat du serveur cible est comparé.
— *ssl_handshake_timeout* est (pour une connexion TLS) le temps en secondes à attendre que la poignée de main TLS se termine avant d'abandonner la connexion. `60.0` secondes si `None` (par défaut).
— *ssl_shutdown_timeout* est le temps en secondes à attendre que l'arrêt SSL se termine avant d'abandonner la connexion. `30.0` secondes si `None` (par défaut).

Ajouté dans la version 3.7.

Modifié dans la version 3.11 : ajout du paramètre *ssl shutdown timeout*

Surveillance de descripteur de fichier

`loop.`**`add_reader`**(*fd*, *callback*, **args*)

Commence à surveiller la disponibilité en lecture du descripteur de fichier *fd* et appelle *callback* avec les arguments spécifiés une fois que *fd* est disponible en lecture.

Any preexisting callback registered for *fd* is cancelled and replaced by *callback*.

`loop.`**`remove_reader`**(*fd*)

Arrête de surveiller le descripteur de fichier *fd* pour la disponibilité en lecture. Renvoie `True` si *fd* était précédemment surveillé pour les lectures.

`loop.`**`add_writer`**(*fd*, *callback*, **args*)

Commence à surveiller le descripteur de fichier *fd* pour la disponibilité en écriture et appelle *callback* avec les arguments spécifiés une fois que *fd* est disponible en écriture.

Any preexisting callback registered for *fd* is cancelled and replaced by *callback*.

Utilisez `functools.partial()` pour *passer des arguments nommés* à *callback*.

`loop.`**`remove_writer`**(*fd*)

Arrête de surveiller le descripteur de fichier *fd* pour la disponibilité en écriture. Renvoie `True` si *fd* était précédemment surveillé pour les écritures.

Voir aussi la section *Prise en charge de la plate-forme* pour certaines limitations de ces méthodes.

Travail direct avec des objets *socket*

En général, les implémentations de protocole qui utilisent des API basées sur le transport telles que `loop.create_connection()` et `loop.create_server()` sont plus rapides que les implémentations qui fonctionnent directement avec les *sockets*. Cependant, il existe des cas d'utilisation où les performances ne sont pas critiques, et travailler directement avec les objets `socket` est plus pratique.

`coroutine` `loop.`**`sock_recv`**(*sock*, *nbytes*)

Reçoit jusqu'à *nbytes* de *sock*. Version asynchrone de `socket.recv()`.

Renvoie les données reçues sous la forme d'un objet bytes.

Le connecteur *sock* ne doit pas être bloquant.

Modifié dans la version 3.7 : même si cette méthode a toujours été documentée en tant que méthode coroutine, les versions antérieures à Python 3.7 renvoyaient un `Future`. Depuis Python 3.7, il s'agit d'une méthode `async def`.

`coroutine` `loop.`**`sock_recv_into`**(*sock*, *buf*)

Reçoit les données de *sock* dans le tampon *buf*. Basée sur le modèle de la méthode bloquante `socket.recv_into()`.

Renvoie le nombre d'octets écrits dans le tampon.

Le connecteur *sock* ne doit pas être bloquant.

Ajouté dans la version 3.7.

coroutine `loop.sock_recvfrom`(*sock*, *bufsize*)

Reçoit un datagramme jusqu'à *bufsize* de *sock*. Version asynchrone de `socket.recvfrom()`.

Renvoie un *n*-uplet (données reçues, adresse distante).

Le connecteur *sock* ne doit pas être bloquant.

Ajouté dans la version 3.11.

coroutine `loop.sock_recvfrom_into`(*sock*, *buf*, *nbytes=0*)

Reçoit un datagramme jusqu'à *nbytes* de *sock* vers *buf*. Version asynchrone de `socket.recvfrom_into()`.

Renvoie un *n*-uplet (nombre d'octets reçus, adresse distante).

Le connecteur *sock* ne doit pas être bloquant.

Ajouté dans la version 3.11.

coroutine `loop.sock_sendall`(*sock*, *data*)

Envoie les données *data* au connecteur *sock*. Version asynchrone de `socket.sendall()`.

Cette méthode continue d'envoyer des données au connecteur jusqu'à ce que toutes les *data* aient été envoyées ou qu'une erreur se produise. `None` est renvoyé en cas de succès. En cas d'erreur, une exception est levée. De plus, il n'existe aucun moyen de déterminer la quantité de données, le cas échéant, qui a été traitée avec succès par l'extrémité réceptrice de la connexion.

Le connecteur *sock* ne doit pas être bloquant.

Modifié dans la version 3.7 : même si la méthode a toujours été documentée en tant que méthode coroutine, avant Python 3.7, elle renvoyait un `Future`. Depuis Python 3.7, il s'agit d'une méthode `async def`.

coroutine `loop.sock_sendto`(*sock*, *data*, *address*)

Envoie un datagramme de *sock* à *address*. Version asynchrone de `socket.sendto()`.

Renvoie le nombre d'octets envoyés.

Le connecteur *sock* ne doit pas être bloquant.

Ajouté dans la version 3.11.

coroutine `loop.sock_connect`(*sock*, *address*)

Connecte *sock* à un connecteur distant situé à *address*.

Version asynchrone de `socket.connect()`.

Le connecteur *sock* ne doit pas être bloquant.

Modifié dans la version 3.5.2 : `address` n'a plus besoin d'être résolu. `sock_connect` essaie de vérifier si *address* est déjà résolue en appelant `socket.inet_pton()`. Sinon, `loop.getaddrinfo()` est utilisé pour résoudre *address*.

> **Voir aussi**
>
> `loop.create_connection()` et `asyncio.open_connection()`.

coroutine `loop.sock_accept`(*sock*)

Accepte une connexion. Basée sur le modèle de la méthode bloquante `socket.accept()`.

Le connecteur doit être lié à une adresse et écouter les connexions. La valeur de retour est une paire `(conn, adresse)` où *conn* est un *nouvel* objet socket utilisable pour envoyer et recevoir des données sur la connexion, et *adresse* est l'adresse liée au connecteur de l'autre côté de la connexion.

Le connecteur *sock* ne doit pas être bloquant.

Modifié dans la version 3.7 : même si la méthode a toujours été documentée en tant que méthode coroutine, avant Python 3.7, elle renvoyait un `Future`. Depuis Python 3.7, il s'agit d'une méthode `async def`.

> **Voir aussi**
>
> `loop.create_server()` et `start_server()`.

coroutine `loop.`**`sock_sendfile`**(*sock*, *file*, *offset=0*, *count=None*, *, *fallback=True*)

Envoie le fichier en utilisant `os.sendfile` (haute performance) si possible. Renvoie le nombre total d'octets envoyés.

Version asynchrone de `socket.sendfile()`.

sock doit être un `socket.SOCK_STREAM` `socket` non bloquant.

file doit être un objet fichier normal ouvert en mode binaire.

offset indique où commencer la lecture du fichier. Si spécifié, *count* est le nombre total d'octets à transmettre, par opposition à l'envoi du fichier jusqu'à ce que EOF soit atteint. La position du fichier est toujours mise à jour, même lorsque cette méthode génère une erreur. `file.tell()` peut être utilisée pour obtenir le nombre d'octets réellement envoyés.

fallback, lorsqu'il est défini à `True`, permet à *asyncio* de lire et d'envoyer manuellement le fichier lorsque la plateforme ne prend pas en charge l'appel système *sendfile* (par exemple, Windows ou connecteur SSL sous Unix).

Lève une `SendfileNotAvailableError` si le système ne prend pas en charge l'appel système *sendfile* et que *fallback* est `False`.

Le connecteur *sock* ne doit pas être bloquant.

Ajouté dans la version 3.7.

DNS

coroutine `loop.`**`getaddrinfo`**(*host*, *port*, *, *family=0*, *type=0*, *proto=0*, *flags=0*)

Version asynchrone de `socket.getaddrinfo()`.

coroutine `loop.`**`getnameinfo`**(*sockaddr*, *flags=0*)

Version asynchrone de `socket.getnameinfo()`.

> **Note**
>
> Both *getaddrinfo* and *getnameinfo* internally utilize their synchronous versions through the loop's default thread pool executor. When this executor is saturated, these methods may experience delays, which higher-level networking libraries may report as increased timeouts. To mitigate this, consider using a custom executor for other user tasks, or setting a default executor with a larger number of workers.

Modifié dans la version 3.7 : les méthodes *getaddrinfo* et *getnameinfo* ont toujours été documentées pour renvoyer une coroutine, mais avant Python 3.7, elles renvoyaient en fait des objets `asyncio.Future`. À partir de Python 3.7, les deux méthodes sont des coroutines.

Travail avec des tubes (*pipes*)

coroutine `loop.`**`connect_read_pipe`**(*protocol_factory*, *pipe*)

Branche l'extrémité en lecture du tube *pipe* à la boucle d'évènements.

protocol_factory doit être un appelable renvoyant un protocole gérant le *protocole asyncio*.

pipe est un *simili-fichier*.

Renvoie la paire `(transport, protocol)`, où *transport* prend en charge l'interface `ReadTransport` et *protocol* est un objet instancié par *protocol_factory*.

Avec la boucle d'événements `SelectorEventLoop`, le *pipe* est mis en mode non bloquant.

coroutine `loop.`**connect_write_pipe**(*protocol_factory*, *pipe*)

Branche l'extrémité en écriture de *pipe* à la boucle d'évènements.

protocol_factory doit être un appelable renvoyant un protocole gérant le *protocole asyncio*.

pipe est un *simili-fichier*.

Renvoie la paire `(transport, protocol)`, où *transport* prend en charge l'interface `WriteTransport` et *protocol* est un objet instancié par *protocol_factory*.

Avec la boucle d'événements `SelectorEventLoop`, le *pipe* est mis en mode non bloquant.

Note

`SelectorEventLoop` ne prend pas en charge les méthodes ci-dessus sous Windows. Utilisez `ProactorEventLoop` à la place pour Windows.

Voir aussi

les méthodes `loop.subprocess_exec()` et `loop.subprocess_shell()`.

Signaux Unix

`loop.`**add_signal_handler**(*signum*, *callback*, **args*)

Définit *callback* comme gestionnaire du signal *signum*.

La fonction de rappel sera appelée par *loop*, avec d'autres rappels en file d'attente et des coroutines exécutables de cette boucle d'événements. Contrairement aux gestionnaires de signaux enregistrés à l'aide de `signal.signal()`, un rappel enregistré avec cette fonction est autorisé à interagir avec la boucle d'événements.

Lève une `ValueError` si le numéro de signal est invalide ou non attrapable. Lève une `RuntimeError` s'il y a un problème lors de la configuration du gestionnaire.

Utilisez `functools.partial()` pour *passer des arguments nommés* à *callback*.

Comme `signal.signal()`, cette fonction doit être invoquée dans le fil d'exécution principal.

`loop.`**remove_signal_handler**(*sig*)

Supprime le gestionnaire du signal *sig*.

Renvoie `True` si le gestionnaire de signal a été supprimé, ou `False` si aucun gestionnaire n'a été défini pour le signal donné.

Availability : Unix.

Voir aussi

le module `signal`.

Exécution de code dans des pools de threads ou de processus

awaitable `loop.`**run_in_executor**(*executor*, *func*, **args*)

Fait en sorte que *func* soit appelée dans l'exécuteur spécifié.

The *executor* argument should be an `concurrent.futures.Executor` instance. The default executor is used if *executor* is `None`. The default executor can be set by `loop.set_default_executor()`, otherwise, a

concurrent.futures.ThreadPoolExecutor will be lazy-initialized and used by *run_in_executor()* if needed.

Exemple :

```
import asyncio
import concurrent.futures

def blocking_io():
    # File operations (such as logging) can block the
    # event loop: run them in a thread pool.
    with open('/dev/urandom', 'rb') as f:
        return f.read(100)

def cpu_bound():
    # CPU-bound operations will block the event loop:
    # in general it is preferable to run them in a
    # process pool.
    return sum(i * i for i in range(10 ** 7))

async def main():
    loop = asyncio.get_running_loop()

    ## Options:

    # 1. Run in the default loop's executor:
    result = await loop.run_in_executor(
        None, blocking_io)
    print('default thread pool', result)

    # 2. Run in a custom thread pool:
    with concurrent.futures.ThreadPoolExecutor() as pool:
        result = await loop.run_in_executor(
            pool, blocking_io)
        print('custom thread pool', result)

    # 3. Run in a custom process pool:
    with concurrent.futures.ProcessPoolExecutor() as pool:
        result = await loop.run_in_executor(
            pool, cpu_bound)
        print('custom process pool', result)

    # 4. Run in a custom interpreter pool:
    with concurrent.futures.InterpreterPoolExecutor() as pool:
        result = await loop.run_in_executor(
            pool, cpu_bound)
        print('custom interpreter pool', result)

if __name__ == '__main__':
    asyncio.run(main())
```

Notez que la garde du point d'entrée (`if __name__ == '__main__'`) est requis pour l'option 3 en raison des particularités de *multiprocessing*, qui est utilisé par *ProcessPoolExecutor*. Voir *Importation sécurisée du module principal*.

Cette méthode renvoie un objet *asyncio.Future*.

Utilisez *functools.partial()* pour *passer des arguments nommés* à *func*.

Modifié dans la version 3.5.3 : *loop.run_in_executor()* ne configure plus le `max_workers` de l'exécuteur

de pool de threads qu'il crée, laissant à la place à l'exécuteur de pool de threads (`ThreadPoolExecutor`) le soin de définir le défaut.

`loop.set_default_executor(executor)`

Set *executor* as the default executor used by `run_in_executor()`. *executor* must be an instance of `ThreadPoolExecutor`, which includes `InterpreterPoolExecutor`.

Modifié dans la version 3.11 : *executor* doit être une instance de `ThreadPoolExecutor`.

API de gestion d'erreur

Permet de personnaliser la façon dont les exceptions sont gérées dans la boucle d'événements.

`loop.set_exception_handler(handler)`

Définit *handler* comme nouveau gestionnaire d'exceptions de boucle d'événements.

Si *handler* est `None`, le gestionnaire d'exceptions par défaut est activé. Sinon, *handler* doit être un appelable avec la signature correspondant à `(loop, context)`, où `loop` est une référence à la boucle d'événements active et `context` est un `dict` contenant les détails de l'exception (voir la documentation `call_exception_handler()` pour plus de détails sur le contexte).

If the handler is called on behalf of a `Task` or `Handle`, it is run in the `contextvars.Context` of that task or callback handle.

Modifié dans la version 3.12 : The handler may be called in the `Context` of the task or handle where the exception originated.

`loop.get_exception_handler()`

Renvoie le gestionnaire d'exceptions actuel ou `None` si aucun gestionnaire d'exceptions personnalisé n'a été défini.

Ajouté dans la version 3.5.2.

`loop.default_exception_handler(context)`

Gestionnaire d'exception par défaut.

Appelée lorsqu'une exception se produit et qu'aucun gestionnaire d'exception n'est défini. Elle peut être appelée par un gestionnaire d'exceptions personnalisé qui souhaite s'en remettre au comportement du gestionnaire par défaut.

Le paramètre *context* a la même signification que dans `call_exception_handler()`.

`loop.call_exception_handler(context)`

Appelle le gestionnaire d'exception de la boucle d'évènements actuelle.

context est un objet `dict` contenant les clés suivantes (de nouvelles clés pourront être introduites dans les futures versions de Python) :
— 'message' : message d'erreur ;
— 'exception' (facultatif) : objet exception ;
— 'future' (facultatif) : instance de `asyncio.Future` ;
— 'task' (facultatif) : instance de `asyncio.Task` ;
— *'handle'* (facultatif) : instance de `asyncio.Handle` ;
— *'protocol'* (facultatif) : instance de *protocole asyncio* ;
— 'transport' (facultatif) : instance de *transport asyncio* ;
— 'socket' (facultatif) : instance de `socket.socket` ;
— ***'asyncgen'* (facultatif) : générateur asynchrone qui a causé** l'exception

> **Note**
>
> This method should not be overloaded in subclassed event loops. For custom exception handling, use the `set_exception_handler()` method.

Activation du mode débogage

`loop.get_debug()`

Obtient le mode de débogage (`bool`) de la boucle d'événements.

La valeur par défaut est `True` si la variable d'environnement `PYTHONASYNCIODEBUG` est définie sur une chaîne non vide, `False` sinon.

`loop.set_debug(enabled : bool)`

Active le mode débogage pour la boucle d'évènements.

Modifié dans la version 3.7 : le nouveau *mode de développement Python* peut désormais également être utilisé pour activer le mode de débogage.

`loop.slow_callback_duration`

This attribute can be used to set the minimum execution duration in seconds that is considered "slow". When debug mode is enabled, "slow" callbacks are logged.

Default value is 100 milliseconds.

Voir aussi

le *mode debogage d'asyncio*.

Exécution de sous-processus

Les méthodes décrites dans ces sous-sections sont de bas niveau. Dans le code *async/await* normal, pensez à utiliser les fonctions de commodité de haut niveau `asyncio.create_subprocess_shell()` et `asyncio.create_subprocess_exec()` à la place.

Note

sous Windows, la boucle d'événements par défaut `ProactorEventLoop` prend en charge les sous-processus, contrairement à `SelectorEventLoop`. Voir *Prise en charge des sous-processus sous Windows* pour plus de détails.

coroutine `loop.subprocess_exec(protocol_factory, *args, stdin=subprocess.PIPE, stdout=subprocess.PIPE, stderr=subprocess.PIPE, **kwargs)`

Crée un sous-processus à partir d'un ou plusieurs arguments de chaîne spécifiés par *args*.

args doit être une liste de chaînes représentée par :
— `str` ;
— ou `bytes`, encodés selon l'*encodage du système de fichiers*.
La première chaîne spécifie l'exécutable du programme et les chaînes restantes spécifient les arguments. Ensemble, les arguments de chaîne forment le `argv` du programme.

C'est similaire à la classe standard de la bibliothèque `subprocess.Popen` appelée avec `shell=False` et la liste des chaînes passées en premier argument ; cependant, où `Popen` prend un seul argument qui est une liste de chaînes, *subprocess_exec* prend plusieurs arguments de chaînes.

Le *protocol_factory* doit être un appelable renvoyant une sous-classe de la classe `asyncio.SubprocessProtocol`.

Autres paramètres :
— *stdin* peut être l'un de ces éléments :
  — a file-like object
  — an existing file descriptor (a positive integer), for example those created with `os.pipe()`
  — la constante `subprocess.PIPE` (par défaut) qui va créer un nouveau tube et le connecter,
  — la valeur `None` qui fera que le sous-processus héritera du descripteur de fichier de ce processus,
  — la constante `subprocess.DEVNULL` qui indique que le fichier spécial `os.devnull` sera utilisé.

— *stdout* peut être l'un de ces éléments :
  — a file-like object
  — la constante `subprocess.PIPE` (par défaut) qui va créer un nouveau tube et le connecter,
  — la valeur `None` qui fera que le sous-processus héritera du descripteur de fichier de ce processus,
  — la constante `subprocess.DEVNULL` qui indique que le fichier spécial `os.devnull` sera utilisé.
— *stderr* peut être l'un de ces éléments :
  — a file-like object
  — la constante `subprocess.PIPE` (par défaut) qui va créer un nouveau tube et le connecter,
  — la valeur `None` qui fera que le sous-processus héritera du descripteur de fichier de ce processus,
  — la constante `subprocess.DEVNULL` qui indique que le fichier spécial `os.devnull` sera utilisé.
  — la constante `subprocess.STDOUT` qui connectera le flux d'erreur standard au flux de sortie standard du processus.
— Tous les autres arguments nommés sont passés à `subprocess.Popen` sans interprétation, à l'exception de *bufsize*, *universal_newlines*, *shell*, *text*, *encoding* et *errors*, qui ne doivent pas être spécifiés du tout.

  L'API de sous-processus `asyncio` ne prend pas en charge le décodage des flux sous forme de texte. `bytes.decode()` peut être utilisée pour convertir les octets renvoyés par le flux en texte.

If a file-like object passed as *stdin*, *stdout* or *stderr* represents a pipe, then the other side of this pipe should be registered with `connect_write_pipe()` or `connect_read_pipe()` for use with the event loop.

Voir le constructeur de la classe `subprocess.Popen` pour la documentation sur les autres arguments.

Renvoie une paire `(transport, protocol)`, où *transport* est conforme à la classe de base `asyncio.SubprocessTransport` et *protocol* est un objet instancié par *protocol_factory*.

coroutine loop.**subprocess_shell**(*protocol_factory*, *cmd*, **, stdin=subprocess.PIPE, stdout=subprocess.PIPE, stderr=subprocess.PIPE, **kwargs*)

Crée un sous-processus à partir de *cmd*, qui peut être une chaîne `str` ou une chaîne `bytes` encodée avec l'*encodage du système de fichiers*, en utilisant la syntaxe "shell" de la plate-forme.

C'est similaire à la classe standard de la bibliothèque `subprocess.Popen` appelée avec `shell=True`.

Le *protocol_factory* doit être un appelable renvoyant une sous-classe de la classe `SubprocessProtocol`.

Voir `subprocess_exec()` pour plus de détails sur les arguments restants.

Renvoie une paire `(transport, protocol)`, où *transport* est conforme à la classe de base `SubprocessTransport` et *protocol* est un objet instancié par *protocol_factory*.

Note

il est de la responsabilité de l'application de s'assurer que tous les espaces blancs et les caractères spéciaux sont correctement échappés pour éviter les vulnérabilités d'injection de shell. La fonction `shlex.quote()` peut être utilisée pour échapper correctement les espaces blancs et les caractères spéciaux dans les chaînes qui seront utilisées pour construire des commandes shell.

Fonctions de rappel sur des descripteurs

class asyncio.**Handle**

Objet encapsulant une fonction de rappel renvoyé par `loop.call_soon()`, `loop.call_soon_threadsafe()`.

get_context()

Return the `contextvars.Context` object associated with the handle.

Ajouté dans la version 3.12.

cancel()

Annule le rappel. Si le rappel a déjà été annulé ou exécuté, cette méthode n'a aucun effet.

cancelled()

Renvoie `True` si la fonction de rappel a été annulée.

Ajouté dans la version 3.7.

class asyncio.**TimerHandle**

Objet encapsulant la fonction de rappel renvoyé par *loop.call_later()* et *loop.call_at()*.

Cette classe est une sous-classe de *Handle*.

when()

Renvoie une heure de rappel planifiée sous forme de *float* secondes.

L'heure est un horodatage absolu, utilisant la même référence de temps que *loop.time()*.

Ajouté dans la version 3.7.

Objets Serveur

Les objets serveur sont créés par les fonctions *loop.create_server()*, *loop.create_unix_server()*, *start_server()* et *start_unix_server()*.

Do not instantiate the *Server* class directly.

class asyncio.**Server**

Les objets *Server* sont des gestionnaires de contexte asynchrones. Lorsqu'il est utilisé dans une instruction `async with`, il est garanti que l'objet Serveur est fermé et n'accepte pas de nouvelle connexion lorsque l'instruction `async with` est terminée :

```
srv = await loop.create_server(...)

async with srv:
    # some code

# At this point, srv is closed and no longer accepts new connections.
```

Modifié dans la version 3.7 : l'objet serveur est un gestionnaire de contexte asynchrone depuis Python 3.7.

Modifié dans la version 3.11 : This class was exposed publicly as `asyncio.Server` in Python 3.9.11, 3.10.3 and 3.11.

close()

Arrête le serveur : ferme les connecteurs d'écoute et définit l'attribut *sockets* à `None`.

Les connecteurs qui représentent les connexions client entrantes existantes restent ouvertes.

The server is closed asynchronously ; use the *wait_closed()* coroutine to wait until the server is closed (and no more connections are active).

close_clients()

Close all existing incoming client connections.

Calls *close()* on all associated transports.

close() should be called before *close_clients()* when closing the server to avoid races with new clients connecting.

Ajouté dans la version 3.13.

abort_clients()

Close all existing incoming client connections immediately, without waiting for pending operations to complete.

Calls *abort()* on all associated transports.

close() should be called before *abort_clients()* when closing the server to avoid races with new clients connecting.

Ajouté dans la version 3.13.

get_loop()

Renvoie la boucle d'événement associée à l'objet serveur.

Ajouté dans la version 3.7.

coroutine start_serving()

Commence à accepter les connexions.

Cette méthode est idempotente, elle peut donc être appelée lorsque le serveur est déjà en service.

Le paramètre nommé *start_serving* de `loop.create_server()` et `asyncio.start_server()` permet de créer un objet *Server* qui n'accepte pas les connexions initialement. Dans ce cas, `Server.start_serving()` ou `Server.serve_forever()` peut être utilisée pour que le serveur commence à accepter les connexions.

Ajouté dans la version 3.7.

coroutine serve_forever()

Commence à accepter les connexions jusqu'à ce que la coroutine soit annulée. L'annulation de la tâche `serve_forever` provoque la fermeture du serveur.

Cette méthode peut être appelée si le serveur accepte déjà les connexions. Une seule tâche `serve_forever` peut exister par objet *Server*.

Exemple :

```
async def client_connected(reader, writer):
    # Communicate with the client with
    # reader/writer streams.  For example:
    await reader.readline()

async def main(host, port):
    srv = await asyncio.start_server(
        client_connected, host, port)
    await srv.serve_forever()

asyncio.run(main('127.0.0.1', 0))
```

Ajouté dans la version 3.7.

is_serving()

Renvoie `True` si le serveur accepte de nouvelles connexions.

Ajouté dans la version 3.7.

coroutine wait_closed()

Wait until the `close()` method completes and all active connections have finished.

sockets

List of socket-like objects, `asyncio.trsock.TransportSocket`, which the server is listening on.

Modifié dans la version 3.7 : avant Python 3.7, `Server.sockets` renvoyait directement une liste interne de sockets de serveur. En 3.7, une copie de cette liste est renvoyée.

Implémentations de boucle d'évènements

asyncio est livré avec deux implémentations de boucles d'événements différentes : `SelectorEventLoop` et `ProactorEventLoop`.

By default asyncio is configured to use `EventLoop`.

class asyncio.**SelectorEventLoop**

A subclass of `AbstractEventLoop` based on the `selectors` module.

Utilise le *sélecteur* le plus efficace disponible pour la plate-forme donnée. Il est également possible de configurer manuellement l'implémentation exacte du sélecteur à utiliser :

```
import asyncio
import selectors

async def main():
    ...

loop_factory = lambda: asyncio.SelectorEventLoop(selectors.SelectSelector())
asyncio.run(main(), loop_factory=loop_factory)
```

Availability : Unix, Windows.

class asyncio.**ProactorEventLoop**

A subclass of `AbstractEventLoop` for Windows that uses "I/O Completion Ports" (IOCP).

Availability : Windows.

> **Voir aussi**
>
> MSDN documentation on I/O Completion Ports.

class asyncio.**EventLoop**

An alias to the most efficient available subclass of `AbstractEventLoop` for the given platform.

It is an alias to `SelectorEventLoop` on Unix and `ProactorEventLoop` on Windows.

Ajouté dans la version 3.13.

class asyncio.**AbstractEventLoop**

Classe mère abstraite pour les boucles d'événements conforme à *asyncio*.

La section *Méthodes de la boucle d'évènements* liste toutes les méthodes qu'une implémentation alternative de `AbstractEventLoop` doit définir.

Exemples

Notez que tous les exemples de cette section montrent **à dessein** comment utiliser les API de boucle d'événement de bas niveau, telles que `loop.run_forever()` et `loop.call_soon()`. Les applications *asyncio* modernes ont rarement besoin d'être écrites de cette façon ; pensez à utiliser les fonctions de haut niveau comme `asyncio.run()`.

"Hello World" avec `call_soon()`

Un exemple utilisant la méthode `loop.call_soon()` pour programmer un rappel. Le rappel affiche `"Hello World"` puis arrête la boucle d'événements :

```
import asyncio

def hello_world(loop):
    """A callback to print 'Hello World' and stop the event loop"""
    print('Hello World')
    loop.stop()

loop = asyncio.new_event_loop()

# Schedule a call to hello_world()
loop.call_soon(hello_world, loop)
```

(suite sur la page suivante)

(suite de la page précédente)

```
# Blocking call interrupted by loop.stop()
try:
    loop.run_forever()
finally:
    loop.close()
```

Voir aussi

un exemple similaire de *Hello World* créé avec une coroutine et la fonction `run()`.

Affichage de la date actuelle avec `call_later()`

Un exemple de rappel affichant la date actuelle toutes les secondes. Le rappel utilise la méthode `loop.call_later()` pour se re-planifier après 5 secondes, puis arrête la boucle d'événements :

```
import asyncio
import datetime

def display_date(end_time, loop):
    print(datetime.datetime.now())
    if (loop.time() + 1.0) < end_time:
        loop.call_later(1, display_date, end_time, loop)
    else:
        loop.stop()

loop = asyncio.new_event_loop()

# Schedule the first call to display_date()
end_time = loop.time() + 5.0
loop.call_soon(display_date, end_time, loop)

# Blocking call interrupted by loop.stop()
try:
    loop.run_forever()
finally:
    loop.close()
```

Voir aussi

un exemple similaire de *date actuelle* créé avec une coroutine et la fonction `run()`.

Surveillance des événements de lecture pour un descripteur de fichier

Attend qu'un descripteur de fichier reçoive des données en utilisant la méthode `loop.add_reader()` puis ferme la boucle d'événements :

```
import asyncio
from socket import socketpair

# Create a pair of connected file descriptors
rsock, wsock = socketpair()

loop = asyncio.new_event_loop()
```

(suite sur la page suivante)

(suite de la page précédente)

```
def reader():
    data = rsock.recv(100)
    print("Received:", data.decode())

    # We are done: unregister the file descriptor
    loop.remove_reader(rsock)

    # Stop the event loop
    loop.stop()

# Register the file descriptor for read event
loop.add_reader(rsock, reader)

# Simulate the reception of data from the network
loop.call_soon(wsock.send, 'abc'.encode())

try:
    # Run the event loop
    loop.run_forever()
finally:
    # We are done. Close sockets and the event loop.
    rsock.close()
    wsock.close()
    loop.close()
```

Voir aussi

— un *exemple* similaire utilisant les transports, les protocoles et la méthode `loop.create_connection()`,
— un autre *exemple* utilisant la fonction et les flux de haut niveau `asyncio.open_connection()`.

Gestion des signaux *SIGINT* et *SIGTERM*

(Cet exemple ne fonctionne que sur Unix.)

Register handlers for signals `SIGINT` and `SIGTERM` using the `loop.add_signal_handler()` method :

```
import asyncio
import functools
import os
import signal

def ask_exit(signame, loop):
    print("got signal %s: exit" % signame)
    loop.stop()

async def main():
    loop = asyncio.get_running_loop()

    for signame in {'SIGINT', 'SIGTERM'}:
        loop.add_signal_handler(
            getattr(signal, signame),
            functools.partial(ask_exit, signame, loop))

    await asyncio.sleep(3600)
```

(suite sur la page suivante)

(suite de la page précédente)

```
print("Event loop running for 1 hour, press Ctrl+C to interrupt.")
print(f"pid {os.getpid()}: send SIGINT or SIGTERM to exit.")

asyncio.run(main())
```

19.1.10 Futures

Code source : Lib/asyncio/futures.py, Lib/asyncio/base_futures.py

---

Les objets *Future* sont utilisés comme passerelles entre du **code bas niveau basé sur des fonctions de rappel** et du code haut niveau utilisant *async* et *await*.

Fonctions pour *Future*

`asyncio.`**`isfuture`**(*obj*)

Renvoie `True` si *obj* est soit :
— une instance de `asyncio.Future`;
— une instance de `asyncio.Task`;
— un objet se comportant comme *Future* et possédant un attribut `_asyncio_future_blocking`.
Ajouté dans la version 3.5.

`asyncio.`**`ensure_future`**(*obj*, *, *loop=None*)

Renvoie :
— l'objet *obj* tel quel si c'est un objet `Future`, `Task` ou se comportant comme un *Future*. `isfuture()` est utilisée pour le test ;
— un objet `Task` encapsulant *obj* si ce dernier est une coroutine (`iscoroutine()` est utilisée pour le test). Dans ce cas, l'exécution de la coroutine sera planifiée par `ensure_future()` ;
— un objet `Task` qui attendra (*await*) *obj* si ce dernier peut être attendu (*awaitable*). `iscoroutine()` est utilisée pour le test.
Si *obj* ne correspond à aucun des critères ci-dessus, une exception `TypeError` est levée.

> **Important**
>
> Voir aussi la fonction `create_task()` qui est la manière privilégiée pour créer des nouvelles tâches.
>
> Gardez une référence au résultat de cette fonction pour éviter de voir une tâche disparaitre au milieu de son exécution.

Modifié dans la version 3.5.1 : La fonction accepte n'importe quel objet *awaitable*.

Obsolète depuis la version 3.10 : Un `DeprecationWarning` est levé si *obj* n'est pas un objet se comportant comme un *Future*, si *loop* n'est pas spécifié et s'il n'y a pas de boucle d'évènements en cours d'exécution.

`asyncio.`**`wrap_future`**(*future*, *, *loop=None*)

Encapsule un objet `concurrent.futures.Future` dans un objet `asyncio.Future`.

Obsolète depuis la version 3.10 : Un `DeprecationWarning` est levé si *future* n'est pas un objet se comportant comme un *Future*, si *loop* n'est pas spécifié et s'il n'y a pas de boucle d'évènements en cours d'exécution.

Objet *Future*

`class` `asyncio.`**`Future`**(*, *loop=None*)

Un *Future* représente le résultat final d'une opération asynchrone. Il n'est pas conçu pour pouvoir être utilisé par plusieurs fils d'exécution.

Future est un objet qui peut être attendu (*awaitable*). Les coroutines peuvent attendre les objets *Future* jusqu'à ce qu'ils renvoient un résultat, ils lèvent une exception ou qu'ils soient annulés. Un *Future* peut être attendu plusieurs fois et le résultat est le même.

Les *Futures* sont habituellement utilisés pour permettre à du code bas niveau basé sur des fonctions de rappel (par exemple : les protocoles utilisant *asyncio transports*) d'interagir avec du code haut niveau utilisant *async* et *await*.

Une bonne règle empirique est de ne jamais exposer des objets *Future* dans des *API* destinées à l'utilisateur. La façon privilégiée de créer des objets *Future* est d'appeler la méthode `loop.create_future()`. Cela permet aux implémentations alternatives de la boucle d'évènements d'utiliser leur propre implémentation de l'objet *Future*.

Modifié dans la version 3.7 : Ajout du support du module `contextvars`.

Obsolète depuis la version 3.10 : Un `DeprecationWarning` est levé si *loop* n'est pas spécifié et s'il n'y a pas de boucle d'évènements en cours d'exécution.

result()

Renvoie le résultat du *Future*.

Si le *Future* est « terminé » et a un résultat défini par la méthode `set_result()`, ce résultat est renvoyé.

Si le *Future* est « terminé » et a une exception définie par la méthode `set_exception()`, cette méthode lève l'exception.

Si le *Future* a été *annulé*, cette méthode lève une exception `CancelledError`.

If the Future's result isn't yet available, this method raises an `InvalidStateError` exception.

set_result(*result*)

Marque le *Future* comme « terminé » et définit son résultat.

Raises an `InvalidStateError` error if the Future is already *done*.

set_exception(*exception*)

Marque le *Future* comme « terminé » et définit une exception.

Raises an `InvalidStateError` error if the Future is already *done*.

done()

Renvoie `True` si le *Future* est « terminé ».

Un *Future* est « terminé » s'il a été « annulé » ou si un résultat ou une exception a été définie par les méthodes `set_result()` ou `set_exception()`.

cancelled()

Renvoie `True` si le *Future* a été « annulé ».

Cette méthode est habituellement utilisée pour vérifier qu'un *Future* n'est pas « annulé » avant de définir un résultat ou une exception pour celui-ci :

```
if not fut.cancelled():
    fut.set_result(42)
```

add_done_callback(*callback*, *, *context=None*)

Ajoute une fonction de rappel à exécuter lorsque le *Future* est « terminé ».

L'argument *callback* est appelé avec l'objet *Future* comme seul argument.

Si le *Future* est déjà « terminé » lorsque la méthode est appelée, l'exécution de la fonction de rappel est planifiée avec `loop.call_soon()`.

L'argument nommé optionnel *context* permet de spécifier une classe `contextvars.Context` personnalisée dans laquelle la fonction de rappel s'exécutera. Le contexte actuel est utilisé si *context* n'est pas fourni.

`functools.partial()` peut être utilisée pour passer des paramètres à la fonction de rappel :

```
# Call 'print("Future:", fut)' when "fut" is done.
fut.add_done_callback(
    functools.partial(print, "Future:"))
```

Modifié dans la version 3.7 : Ajout de l'argument nommé *context*. Voir **PEP 567** pour plus de détails.

remove_done_callback(*callback*)

Retire *callback* de la liste de fonctions de rappel.

Renvoie le nombre de fonctions de rappel retiré. La méthode renvoie généralement 1, à moins que la fonction ait été ajoutée plus d'une fois.

cancel(*msg=None*)

Annule le *Future* et planifie l'exécution des fonctions de rappel.

Si le *Future* est déjà « terminé » ou « annulé », renvoie `False`. Autrement, change l'état du *Future* à « annulé », planifie l'exécution des fonctions de rappel et renvoie `True`.

Modifié dans la version 3.9 : Ajout du paramètre *msg*.

exception()

Renvoie l'exception définie pour ce *Future*.

L'exception, ou `None` si aucune exception n'a été définie, est renvoyé seulement si le *Future* est « terminé ».

Si le *Future* a été *annulé*, cette méthode lève une exception `CancelledError`.

Si le *Future* n'est pas encore « terminé », cette méthode lève une exception `InvalidStateError`.

get_loop()

Renvoie la boucle d'évènements à laquelle le *Future* est attaché.

Ajouté dans la version 3.7.

Cet exemple crée un objet *Future*, puis crée et planifie l'exécution d'une tâche asynchrone qui définira le résultat du *Future* et attend jusqu'à ce que le *Future* ait un résultat :

```
async def set_after(fut, delay, value):
    # Sleep for *delay* seconds.
    await asyncio.sleep(delay)

    # Set *value* as a result of *fut* Future.
    fut.set_result(value)

async def main():
    # Get the current event loop.
    loop = asyncio.get_running_loop()

    # Create a new Future object.
    fut = loop.create_future()

    # Run "set_after()" coroutine in a parallel Task.
    # We are using the low-level "loop.create_task()" API here because
    # we already have a reference to the event loop at hand.
    # Otherwise we could have just used "asyncio.create_task()".
    loop.create_task(
        set_after(fut, 1, '... world'))

    print('hello ...')

    # Wait until *fut* has a result (1 second) and print it.
    print(await fut)
```

(suite sur la page suivante)

(suite de la page précédente)

```
asyncio.run(main())
```

Important

L'objet *Future* est conçu pour imiter la classe `concurrent.futures.Future`. Les principales différences sont :

- contrairement au *Future asyncio*, les instances de `concurrent.futures.Future` ne peuvent pas être attendues ;
- `asyncio.Future.result()` et `asyncio.Future.exception()` n'acceptent pas d'argument *timeout* ;
- `asyncio.Future.result()` et `asyncio.Future.exception()` lèvent une exception `InvalidStateError` lorsque le *Future* n'est pas « terminé » ;
- les fonctions de rappel enregistrées à l'aide de `asyncio.Future.add_done_callback()` ne sont pas exécutées immédiatement mais planifiées avec `loop.call_soon()` ;
- les *Future asyncio* ne sont pas compatibles avec les fonctions `concurrent.futures.wait()` et `concurrent.futures.as_completed()` ;
- `asyncio.Future.cancel()` accepts an optional `msg` argument, but `concurrent.futures.Future.cancel()` does not.

19.1.11 Transports et Protocoles

Préface

Les transports et les protocoles sont utilisés par les API de boucle d'événements **de bas niveau** telles que `loop.create_connection()`. Ils utilisent un style de programmation basé sur les fonctions de rappel et permettent des implémentations hautes performances de protocoles réseau (par exemple, HTTP) ou de communication inter-processus (*IPC*).

Avant tout, les transports et les protocoles ne doivent être utilisés que dans des bibliothèques et des cadriciels et jamais dans des applications asynchrones de haut niveau.

Cette page de documentation couvre à la fois *Transports* et *Protocoles*.

Introduction

Au plus haut niveau, le transport concerne *comment* les octets sont transmis, tandis que le protocole détermine *quels* octets transmettre (et dans une certaine mesure quand).

Pour l'écrire autrement : un transport est une abstraction pour un connecteur (*socket* ou tout autre point de terminaison d'entrée-sortie) tandis qu'un protocole est une abstraction pour une application, du point de vue du transport.

Encore une autre vue est que les interfaces de transport et de protocole définissent ensemble une interface abstraite pour utiliser les entrées-sorties réseau et les entrées-sorties inter-processus.

Il existe toujours une relation 1 :1 entre les objets de transport et de protocole : le protocole appelle les méthodes de transport pour envoyer des données, tandis que le transport appelle les méthodes de protocole pour lui transmettre les données reçues.

La plupart des méthodes de boucles d'événements orientées connexion (telles que `loop.create_connection()`) acceptent généralement un argument *protocol_factory* utilisé pour créer un objet *Protocol* pour une connexion acceptée, représentée par un objet *Transport*. De telles méthodes renvoient généralement un *n*-uplet `(transport, protocol)`.

Sommaire

Cette page de documentation contient les sections suivantes :

- La section *Transports* documente les classes *asyncio* `BaseTransport`, `ReadTransport`, `WriteTransport`, `Transport`, `DatagramTransport` et `SubprocessTransport`.

— La section *Protocols* documente les classes *asyncio* `BaseProtocol`, `Protocol`, `BufferedProtocol`, `DatagramProtocol` et `SubprocessProtocol`.
— La section *Examples* montre comment utiliser les transports, les protocoles et les API de boucle d'événements de bas niveau.

Transports

Code source : Lib/asyncio/transports.py

Les transports sont des classes fournies par `asyncio` afin d'abstraire différents types de canaux de communication.

Les objets de transport sont toujours instanciés par une *boucle d'événements asyncio*.

asyncio implémente les transports pour TCP, UDP, SSL et les tubes de sous-processus. Les méthodes disponibles sur un transport dépendent du type de transport.

Les classes de transport ne sont *pas compatibles avec les fils d'exécution multiples*.

Hiérarchie des transports

class asyncio.**BaseTransport**

Classe de base pour tous les transports. Contient des méthodes partagées par tous les transports *asyncio*.

class asyncio.**WriteTransport**(*BaseTransport*)

Transport de base pour les connexions en écriture seule.

Les instances de la classe *WriteTransport* sont renvoyées par la méthode de boucle d'événements `loop.connect_write_pipe()` et sont également utilisées par les méthodes liées aux sous-processus comme `loop.subprocess_exec()`.

class asyncio.**ReadTransport**(*BaseTransport*)

Transport de base pour les connexions en lecture seule.

Les instances de la classe *ReadTransport* sont renvoyées par la méthode de boucle d'événements `loop.connect_read_pipe()` et sont également utilisées par les méthodes liées aux sous-processus comme `loop.subprocess_exec()`.

class asyncio.**Transport**(*WriteTransport*, *ReadTransport*)

Interface représentant un transport bidirectionnel, comme une connexion TCP.

L'utilisateur n'instancie pas un transport directement ; il appelle une fonction utilitaire, lui transmettant une fabrique de protocoles et d'autres informations nécessaires pour créer le transport et le protocole.

Les instances de la classe *Transport* sont renvoyées ou utilisées par des méthodes de boucle d'événements comme `loop.create_connection()`, `loop.create_unix_connection()`, `loop.create_server()`, `loop.sendfile()`, etc.

class asyncio.**DatagramTransport**(*BaseTransport*)

Transport pour les connexions par datagrammes (UDP).

Les instances de la classe *DatagramTransport* sont renvoyées par la méthode de boucle d'événements `loop.create_datagram_endpoint()`.

class asyncio.**SubprocessTransport**(*BaseTransport*)

Abstraction pour représenter une connexion entre un parent et son processus enfant au niveau du système d'exploitation.

Les instances de la classe *SubprocessTransport* sont renvoyées par les méthodes de boucle d'événements `loop.subprocess_shell()` et `loop.subprocess_exec()`.

Classe de base des Transports

`BaseTransport.close()`

Ferme le transport.

Si le transport a une mémoire tampon pour les données sortantes, les données mises en mémoire tampon seront vidées de manière asynchrone. Aucune autre donnée ne sera reçue. Une fois que toutes les données mises en mémoire tampon ont été vidées, la méthode `protocol.connection_lost()` sera appelée avec `None` comme argument. Le transport ne doit pas être utilisé une fois qu'il est fermé.

`BaseTransport.is_closing()`

Renvoie `True` si le transport se ferme ou est fermé.

`BaseTransport.get_extra_info(name, default=None)`

Renvoie des informations sur le transport ou les ressources sous-jacentes qu'il utilise.

name est une chaîne représentant l'information spécifique au transport à obtenir.

default est la valeur à renvoyer si les informations ne sont pas disponibles ou si le transport ne prend pas en charge l'implémentation de boucle d'événements tierce donnée ou la plateforme actuelle.

Par exemple, le code suivant tente d'obtenir l'objet *socket* sous-jacent du transport :

```
sock = transport.get_extra_info('socket')
if sock is not None:
    print(sock.getsockopt(...))
```

Catégories d'informations pouvant être interrogées sur certains transports :

- *socket* :
  - `'peername'` : l'adresse distante à laquelle le *socket* est connecté, résultat de `socket.socket.getpeername()` (`None` en cas d'erreur)
  - `'socket'` : Instance de `socket.socket`
  - `'sockname'` : la propre adresse du connecteur, résultat de `socket.socket.getsockname()`
- Connecteur (*socket*) SSL :
  - `'compression'` : l'algorithme de compression utilisé (chaîne de caractères), ou `None` si la connexion n'est pas compressée ; résultat de `ssl.SSLSocket.compression()`
  - `'cipher'` : un *n*-uplet à trois valeurs contenant le nom du chiffrement utilisé, la version du protocole SSL qui définit son utilisation et le nombre de bits secrets utilisés ; résultat de `ssl.SSLSocket.cipher()`
  - `'peercert'` : certificat du pair ; résultat de `ssl.SSLSocket.getpeercert()`
  - `sslcontext'` : instance de `ssl.SSLContext`
  - `'ssl_object'` : instance de `ssl.SSLObject` ou de `ssl.SSLSocket`
- tube :
  - `'pipe'` : objet *pipe*
- sous-processus :
  - `'sous-processus'` : instance de `subprocess.Popen`

`BaseTransport.set_protocol(protocol)`

Change le protocole.

La commutation de protocole ne doit être effectuée que lorsque les deux protocoles sont documentés pour prendre en charge la commutation.

`BaseTransport.get_protocol()`

Renvoie le protocole courant.

Transports en lecture seule

`ReadTransport.is_reading()`

Renvoie `True` si le transport reçoit de nouvelles données.

Ajouté dans la version 3.7.

`ReadTransport.pause_reading()`

Met en pause l'extrémité de réception du transport. Aucune donnée ne sera transmise à la méthode `protocol.data_received()` du protocole jusqu'à ce que `resume_reading()` soit appelée.

Modifié dans la version 3.7 : la méthode est idempotente, c'est-à-dire qu'elle peut être appelée lorsque le transport est déjà en pause ou fermé.

`ReadTransport.resume_reading()`

Reprend la réception. La méthode `protocol.data_received()` du protocole sera appelée à nouveau si certaines données sont disponibles pour la lecture.

Modifié dans la version 3.7 : la méthode est idempotente, c'est-à-dire qu'elle peut être appelée alors que le transport est déjà en train de lire.

Transports en lecture-écriture

`WriteTransport.abort()`

Ferme le transport immédiatement, sans attendre la fin des opérations en attente. Les données mises en mémoire tampon sont perdues. Aucune autre donnée ne sera reçue. La méthode `protocol.connection_lost()` du protocole sera éventuellement appelée avec `None` comme argument.

`WriteTransport.can_write_eof()`

Renvoie `True` si le transport gère `write_eof()`, `False` sinon.

`WriteTransport.get_write_buffer_size()`

Renvoie la taille actuelle du tampon de sortie utilisé par le transport.

`WriteTransport.get_write_buffer_limits()`

Obtient les seuils *high* et *low* pour le contrôle du flux d'écriture. Renvoie un *n*-uplet `(low, high)` où *low* et *high* sont des nombres positifs d'octets.

Utilisez `set_write_buffer_limits()` pour définir les limites.

Ajouté dans la version 3.4.2.

`WriteTransport.set_write_buffer_limits(high=None, low=None)`

Définit les seuils *high* et *low* pour le contrôle du flux d'écriture.

Ces deux valeurs (mesurées en nombre d'octets) contrôlent quand les méthodes `protocol.pause_writing()` et `protocol.resume_writing()` du protocole sont appelées. S'il est spécifié, le seuil bas doit être inférieur ou égal au seuil haut. Ni *high* ni *low* ne peuvent être négatifs.

`pause_writing()` est appelée lorsque la taille du tampon devient supérieure ou égale à la valeur *high*. Si l'écriture a été interrompue, `resume_writing()` est appelée lorsque la taille du tampon devient inférieure ou égale à la valeur *low*.

Les valeurs par défaut sont spécifiques à l'implémentation. Si seul le seuil supérieur est donné, le seuil bas prend par défaut une valeur spécifique à l'implémentation inférieure ou égale au seuil supérieur. Définir *high* sur zéro force également *low* sur zéro et provoque l'appel de `pause_writing()` chaque fois que le tampon devient non vide. Définir *low* sur zéro entraîne l'appel de `resume_writing()` uniquement une fois que le tampon est vide. L'utilisation de zéro pour l'un ou l'autre seuil est généralement sous-optimal car cela réduit les possibilités d'effectuer des entrées-sorties et des calculs simultanément.

Utilisez `get_write_buffer_limits()` pour obtenir les limites.

`WriteTransport.write(data)`

Écrit des octets de *data* sur le transport.

Cette méthode ne bloque pas ; elle met les données en mémoire tampon et s'arrange pour qu'elles soient envoyées de manière asynchrone.

`WriteTransport.writelines(list_of_data)`

Écrit une liste (ou tout itérable) d'octets de données dans le transport. Ceci est fonctionnellement équivalent à appeler `write()` sur chaque élément produit par l'itérable, mais peut être implémentée plus efficacement.

`WriteTransport.write_eof()`

Ferme l'extrémité d'écriture du transport après avoir vidé toutes les données mises en mémoire tampon. Des données peuvent encore être reçues.

Cette méthode peut lever `NotImplementedError` si le transport (par exemple SSL) ne prend pas en charge les connexions semi-fermées.

Transports par datagrammes

`DatagramTransport.sendto(data, addr=None)`

Envoie les octets *data* au pair distant indiqué par *addr* (une adresse cible dépendante du transport). Si *addr* est `None`, les données sont envoyées à l'adresse cible indiquée lors de la création du transport.

Cette méthode ne bloque pas ; elle met les données en mémoire tampon et s'arrange pour qu'elles soient envoyées de manière asynchrone.

Modifié dans la version 3.13 : This method can be called with an empty bytes object to send a zero-length datagram. The buffer size calculation used for flow control is also updated to account for the datagram header.

`DatagramTransport.abort()`

Ferme le transport immédiatement, sans attendre la fin des opérations en attente. Les données mises en mémoire tampon sont perdues. Aucune autre donnée ne sera reçue. La méthode `protocol.connection_lost()` du protocole sera éventuellement appelée avec `None` comme argument.

Transports entre sous-processus

`SubprocessTransport.get_pid()`

Renvoie l'identifiant du sous processus sous la forme d'un nombre entier.

`SubprocessTransport.get_pipe_transport(fd)`

Renvoie le transport pour le canal de communication correspondant au descripteur de fichier *fd* donné sous forme d'un entier :

- `0` : transport de flux en lecture de l'entrée standard (*stdin*), ou `None` si le sous-processus n'a pas été créé avec `stdin=PIPE`
- `1` : transport de flux en écriture de la sortie standard (*stdout*), ou `None` si le sous-processus n'a pas été créé avec `stdout=PIPE`
- `2` : transport de flux en écriture de l'erreur standard (*stderr*), ou `None` si le sous-processus n'a pas été créé avec `stderr=PIPE`
- autre *fd* : `None`

`SubprocessTransport.get_returncode()`

Renvoie le code de retour du sous-processus sous la forme d'un entier ou `None` s'il n'a pas été renvoyé, ce qui est similaire à l'attribut `subprocess.Popen.returncode`.

`SubprocessTransport.kill()`

Tue le sous-processus.

Sur les systèmes POSIX, la fonction envoie SIGKILL au sous-processus. Sous Windows, cette méthode est un alias pour `terminate()`.

Voir aussi `subprocess.Popen.kill()`.

`SubprocessTransport.send_signal(signal)`

Envoie le numéro de *signal* au sous-processus, comme dans `subprocess.Popen.send_signal()`.

`SubprocessTransport.terminate()`

Termine le sous-processus.

On POSIX systems, this method sends `SIGTERM` to the subprocess. On Windows, the Windows API function `TerminateProcess()` is called to stop the subprocess.

Voir aussi `subprocess.Popen.terminate()`.

`SubprocessTransport.close()`

Tue le sous-processus en appelant la méthode `kill()`.

Si le sous-processus n'est pas encore terminé, ferme les transports des tubes *stdin*, *stdout* et *stderr*.

Protocoles

Code source : Lib/asyncio/protocols.py

---

asyncio fournit un ensemble de classes mères abstraites qui doivent être utilisées pour implémenter des protocoles réseau. Ces classes sont destinées à être utilisées avec les *transports*.

Les sous-classes des classes mères abstraites de protocole peuvent implémenter certaines ou toutes les méthodes. Toutes ces méthodes sont des rappels : elles sont appelées par des transports sur certains événements, par exemple lors de la réception de certaines données. Une méthode de protocole de base doit être appelée par le transport correspondant.

Protocoles de base

class `asyncio.BaseProtocol`

Protocole de base avec des méthodes partagées par tous les protocoles.

class `asyncio.Protocol`(*BaseProtocol*)

Classe mère pour l'implémentation des protocoles de streaming (TCP, sockets Unix, etc.).

class `asyncio.BufferedProtocol`(*BaseProtocol*)

Classe mère pour implémenter des protocoles de streaming avec contrôle manuel du tampon de réception.

class `asyncio.DatagramProtocol`(*BaseProtocol*)

Classe mère pour l'implémentation des protocoles par datagrammes (UDP).

class `asyncio.SubprocessProtocol`(*BaseProtocol*)

Classe mère pour implémenter des protocoles communiquant avec des processus enfants (canaux unidirectionnels).

Protocoles de base

Tous les protocoles asynchrones peuvent implémenter des rappels pour les protocoles de base.

Rappels pour les connexions

Les méthodes de rappel pour les connexions concernent tous les protocoles, exactement une fois par connexion réussie. Tous les autres rappels de protocole ne peuvent être appelés qu'entre ces deux méthodes.

`BaseProtocol.connection_made`(*transport*)

Appelée lorsqu'une connexion est établie.

L'argument *transport* est le transport représentant la connexion. Le protocole est chargé de stocker la référence à son transport.

`BaseProtocol.connection_lost`(*exc*)

Appelée lorsqu'une connexion est perdue ou fermée.

L'argument est soit un objet exception soit `None`. Ce dernier signifie qu'un EOF régulier est reçu, ou que la connexion a été interrompue ou fermée par ce côté de la connexion.

Rappels pour le contrôle de flux

Les méthodes de rappel pour le contrôle de flux concernent les transports et sont utilisés pour suspendre ou reprendre l'écriture effectuée par le protocole.

Voir la documentation de la méthode *set_write_buffer_limits()* pour plus de détails.

BaseProtocol.**pause_writing**()

Appelée lorsque la mémoire tampon du transport dépasse la limite supérieure.

BaseProtocol.**resume_writing**()

Appelée lorsque la mémoire tampon du transport passe sous le seuil bas.

Si la taille du tampon est égale au seuil haut, *pause_writing()* n'est pas appelée : la taille du tampon doit être strictement supérieure.

Inversement, *resume_writing()* est appelée lorsque la taille du tampon est égale ou inférieure au seuil bas. Ces conditions de fin sont importantes pour s'assurer que les choses se déroulent comme prévu lorsque l'un ou l'autre seuil est à zéro.

Protocoles connectés

Les méthodes d'événement, telles que *loop.create_server()*, *loop.create_unix_server()*, *loop.create_connection()*, *loop.create_unix_connection()*, *loop.connect_accepted_socket()*, *loop.connect_read_pipe()* et *loop.connect_write_pipe()* acceptent les fabriques qui renvoient des protocoles connectés.

Protocol.**data_received**(*data*)

Appelée lorsque certaines données sont reçues. *data* est un objet bytes non vide contenant les données entrantes.

Le fait que les données soient mises en mémoire tampon, fragmentées ou réassemblées dépend du transport. En général, vous ne devez pas vous fier à une sémantique spécifique et plutôt rendre votre analyse générique et flexible. Cependant, les données sont toujours reçues dans le bon ordre.

La méthode peut être appelée un nombre arbitraire de fois lorsqu'une connexion est ouverte.

Cependant, *protocol.eof_received()* est appelée au plus une fois. Une fois que `eof_received()` est appelée, `data_received()` n'est plus appelée.

Protocol.**eof_received**()

Appelée lorsque l'autre extrémité signale qu'il n'enverra plus de données (par exemple en appelant *transport.write_eof()*, si l'autre extrémité utilise également *asyncio*).

Cette méthode peut renvoyer une valeur évaluée à faux (y compris `None`), auquel cas le transport se ferme de lui-même. À l'inverse, si cette méthode renvoie une valeur évaluée à vrai, le protocole utilisé détermine s'il faut fermer le transport. Puisque l'implémentation par défaut renvoie `None`, elle ferme implicitement la connexion.

Certains transports, y compris SSL, ne prennent pas en charge les connexions semi-fermées, auquel cas renvoyer *True* à partir de cette méthode entraîne la fermeture de la connexion.

Machine à états :

```
start -> connection_made
    [-> data_received]*
    [-> eof_received]?
-> connection_lost -> end
```

Protocoles connectés avec tampon

Ajouté dans la version 3.7.

Les protocoles avec mise en mémoire tampon peuvent être utilisés avec n'importe quelle méthode de boucle d'événements prenant en charge les *protocoles connectés*.

Les implémentations de `BufferedProtocol` permettent une allocation et un contrôle manuels explicites du tampon de réception. Les boucles d'événements peuvent alors utiliser le tampon fourni par le protocole pour éviter les copies de données inutiles. Cela peut entraîner une amélioration notable des performances pour les protocoles qui reçoivent de grandes quantités de données. Des implémentations de protocole sophistiquées peuvent réduire considérablement le nombre d'allocations de mémoire tampon.

Les méthodes de rappel suivantes sont appelées sur les instances `BufferedProtocol` :

`BufferedProtocol.`**`get_buffer`**`(`*sizehint*`)`

Appelée pour allouer un nouveau tampon de réception.

sizehint est la taille minimale recommandée pour le tampon renvoyé. Il est acceptable de renvoyer des tampons plus petits ou plus grands que ce que suggère *sizehint*. Lorsqu'il est défini à −1, la taille du tampon peut être arbitraire. C'est une erreur de renvoyer un tampon de taille nulle.

`get_buffer()` doit renvoyer un objet implémentant le protocole tampon.

`BufferedProtocol.`**`buffer_updated`**`(`*nbytes*`)`

Appelée lorsque le tampon a été mis à jour avec les données reçues.

nbytes est le nombre total d'octets qui ont été écrits dans la mémoire tampon.

`BufferedProtocol.`**`eof_received`**`()`

Voir la documentation de la méthode `protocol.eof_received()`.

`get_buffer()` peut être appelée un nombre arbitraire de fois pendant une connexion. Cependant, `protocol.eof_received()` est appelée au plus une fois et, si elle est appelée, `get_buffer()` et `buffer_updated()` ne seront pas appelées après.

Machine à états :

```
start -> connection_made
    [-> get_buffer
        [-> buffer_updated]?
    ]*
    [-> eof_received]?
-> connection_lost -> end
```

Protocoles par datagrammes (non connectés)

Les instances du protocole par datagrammes doivent être construites par des fabriques de protocole transmises à la méthode `loop.create_datagram_endpoint()`.

`DatagramProtocol.`**`datagram_received`**`(`*data*, *addr*`)`

Appelée lorsqu'un datagramme est reçu. *data* est un objet bytes contenant les données entrantes. *addr* est l'adresse du pair qui envoie les données ; le format exact dépend du transport.

`DatagramProtocol.`**`error_received`**`(`*exc*`)`

Appelée lorsqu'une opération d'envoi ou de réception précédente lève une `OSError`. *exc* est l'instance `OSError`.

Cette méthode est appelée dans de rares cas, lorsque le transport (par exemple UDP) détecte qu'un datagramme n'a pas pu être livré à son destinataire. Cependant, il est courant que les datagrammes qui ne peuvent être acheminés soient supprimés silencieusement.

Note

Sur les systèmes BSD (macOS, FreeBSD, etc.), le contrôle de flux n'est pas pris en charge pour les protocoles par datagrammes, car il n'existe aucun moyen fiable de détecter les échecs d'envoi causés par l'écriture d'un trop grand nombre de paquets.

Le connecteur apparaît toujours « prêt » et les paquets en excès sont supprimés. Une `OSError` avec `errno` définie sur `errno.ENOBUFS` peut être levée ou non ; si elle est levée, c'est communiqué à `DatagramProtocol.error_received()` et ignoré dans le cas contraire.

Protocoles liés aux sous-processus

Les instances de protocole de sous-processus doivent être construites par des fabriques de protocole transmises aux méthodes `loop.subprocess_exec()` et `loop.subprocess_shell()`.

`SubprocessProtocol.`**`pipe_data_received`**(*fd*, *data*)

Appelée lorsqu'un processus enfant écrit sur sa sortie d'erreur ou sa sortie standard.

fd est l'entier représentant le descripteur de fichier du tube.

data est un objet bytes non vide contenant les données reçues.

`SubprocessProtocol.`**`pipe_connection_lost`**(*fd*, *exc*)

Appelé lorsqu'un des tubes de communication avec un sous-processus est fermé.

fd est l'entier représentant le descripteur de fichier qui a été fermé.

`SubprocessProtocol.`**`process_exited`**()

Appelée lorsqu'un processus enfant se termine.

It can be called before `pipe_data_received()` and `pipe_connection_lost()` methods.

Exemples

Serveur *écho* en TCP

Crée un serveur d'écho TCP en utilisant la méthode `loop.create_server()`, renvoie les données reçues et ferme la connexion :

```
import asyncio


class EchoServerProtocol(asyncio.Protocol):
    def connection_made(self, transport):
        peername = transport.get_extra_info('peername')
        print('Connection from {}'.format(peername))
        self.transport = transport

    def data_received(self, data):
        message = data.decode()
        print('Data received: {!r}'.format(message))

        print('Send: {!r}'.format(message))
        self.transport.write(data)

        print('Close the client socket')
        self.transport.close()


async def main():
    # Get a reference to the event loop as we plan to use
    # low-level APIs.
    loop = asyncio.get_running_loop()

    server = await loop.create_server(
```

(suite sur la page suivante)

(suite de la page précédente)

```
        EchoServerProtocol,
        '127.0.0.1', 8888)

    async with server:
        await server.serve_forever()


asyncio.run(main())
```

Voir aussi

L'exemple *Serveur d'écho TCP utilisant des flux* utilise la fonction de haut niveau `asyncio.start_server()`.

Client *écho* en TCP

Client d'écho TCP utilisant la méthode `loop.create_connection()` envoie des données et attend que la connexion soit fermée :

```
import asyncio


class EchoClientProtocol(asyncio.Protocol):
    def __init__(self, message, on_con_lost):
        self.message = message
        self.on_con_lost = on_con_lost

    def connection_made(self, transport):
        transport.write(self.message.encode())
        print('Data sent: {!r}'.format(self.message))

    def data_received(self, data):
        print('Data received: {!r}'.format(data.decode()))

    def connection_lost(self, exc):
        print('The server closed the connection')
        self.on_con_lost.set_result(True)


async def main():
    # Get a reference to the event loop as we plan to use
    # low-level APIs.
    loop = asyncio.get_running_loop()

    on_con_lost = loop.create_future()
    message = 'Hello World!'

    transport, protocol = await loop.create_connection(
        lambda: EchoClientProtocol(message, on_con_lost),
        '127.0.0.1', 8888)

    # Wait until the protocol signals that the connection
    # is lost and close the transport.
    try:
        await on_con_lost
    finally:
```

(suite sur la page suivante)

(suite de la page précédente)

```
        transport.close()


asyncio.run(main())
```

Voir aussi

L'exemple *Client d'écho TCP utilisant des flux* utilise la fonction de haut niveau `asyncio.open_connection()`.

Serveur *écho* en UDP

Serveur d'écho UDP, utilisant la méthode `loop.create_datagram_endpoint()`, qui renvoie les données reçues :

```
import asyncio


class EchoServerProtocol:
    def connection_made(self, transport):
        self.transport = transport

    def datagram_received(self, data, addr):
        message = data.decode()
        print('Received %r from %s' % (message, addr))
        print('Send %r to %s' % (message, addr))
        self.transport.sendto(data, addr)


async def main():
    print("Starting UDP server")

    # Get a reference to the event loop as we plan to use
    # low-level APIs.
    loop = asyncio.get_running_loop()

    # One protocol instance will be created to serve all
    # client requests.
    transport, protocol = await loop.create_datagram_endpoint(
        EchoServerProtocol,
        local_addr=('127.0.0.1', 9999))

    try:
        await asyncio.sleep(3600)  # Serve for 1 hour.
    finally:
        transport.close()


asyncio.run(main())
```

Client *écho* en UDP

Client d'écho UDP, utilisant la méthode `loop.create_datagram_endpoint()`, qui envoie des données et ferme le transport lorsqu'il reçoit la réponse :

```
import asyncio


class EchoClientProtocol:
    def __init__(self, message, on_con_lost):
        self.message = message
        self.on_con_lost = on_con_lost
        self.transport = None

    def connection_made(self, transport):
        self.transport = transport
        print('Send:', self.message)
        self.transport.sendto(self.message.encode())

    def datagram_received(self, data, addr):
        print("Received:", data.decode())

        print("Close the socket")
        self.transport.close()

    def error_received(self, exc):
        print('Error received:', exc)

    def connection_lost(self, exc):
        print("Connection closed")
        self.on_con_lost.set_result(True)


async def main():
    # Get a reference to the event loop as we plan to use
    # low-level APIs.
    loop = asyncio.get_running_loop()

    on_con_lost = loop.create_future()
    message = "Hello World!"

    transport, protocol = await loop.create_datagram_endpoint(
        lambda: EchoClientProtocol(message, on_con_lost),
        remote_addr=('127.0.0.1', 9999))

    try:
        await on_con_lost
    finally:
        transport.close()


asyncio.run(main())
```

Connexion de connecteurs existants

Attend qu'un connecteur reçoive des données en utilisant la méthode `loop.create_connection()` avec un protocole :

```
import asyncio
import socket
```

(suite sur la page suivante)

(suite de la page précédente)

```
class MyProtocol(asyncio.Protocol):

    def __init__(self, on_con_lost):
        self.transport = None
        self.on_con_lost = on_con_lost

    def connection_made(self, transport):
        self.transport = transport

    def data_received(self, data):
        print("Received:", data.decode())

        # We are done: close the transport;
        # connection_lost() will be called automatically.
        self.transport.close()

    def connection_lost(self, exc):
        # The socket has been closed
        self.on_con_lost.set_result(True)


async def main():
    # Get a reference to the event loop as we plan to use
    # low-level APIs.
    loop = asyncio.get_running_loop()
    on_con_lost = loop.create_future()

    # Create a pair of connected sockets
    rsock, wsock = socket.socketpair()

    # Register the socket to wait for data.
    transport, protocol = await loop.create_connection(
        lambda: MyProtocol(on_con_lost), sock=rsock)

    # Simulate the reception of data from the network.
    loop.call_soon(wsock.send, 'abc'.encode())

    try:
        await protocol.on_con_lost
    finally:
        transport.close()
        wsock.close()

asyncio.run(main())
```

Voir aussi

L'exemple *Surveillance des événements de lecture pour un descripteur de fichier* utilise la méthode de bas niveau `loop.add_reader()` pour enregistrer un descripteur de fichier.

L'exemple *Ouverture d'un connecteur pour attendre les données à l'aide de flux* utilise des flux de haut niveau créés par la fonction `open_connection()` dans une coroutine.

loop.subprocess_exec() et *SubprocessProtocol*

Un exemple de protocole de sous-processus utilisé pour obtenir la sortie d'un sous-processus et attendre la sortie du sous-processus.

Le sous-processus est créé par la méthode `loop.subprocess_exec()` :

```
import asyncio
import sys

class DateProtocol(asyncio.SubprocessProtocol):
    def __init__(self, exit_future):
        self.exit_future = exit_future
        self.output = bytearray()
        self.pipe_closed = False
        self.exited = False

    def pipe_connection_lost(self, fd, exc):
        self.pipe_closed = True
        self.check_for_exit()

    def pipe_data_received(self, fd, data):
        self.output.extend(data)

    def process_exited(self):
        self.exited = True
        # process_exited() method can be called before
        # pipe_connection_lost() method: wait until both methods are
        # called.
        self.check_for_exit()

    def check_for_exit(self):
        if self.pipe_closed and self.exited:
            self.exit_future.set_result(True)

async def get_date():
    # Get a reference to the event loop as we plan to use
    # low-level APIs.
    loop = asyncio.get_running_loop()

    code = 'import datetime; print(datetime.datetime.now())'
    exit_future = asyncio.Future(loop=loop)

    # Create the subprocess controlled by DateProtocol;
    # redirect the standard output into a pipe.
    transport, protocol = await loop.subprocess_exec(
        lambda: DateProtocol(exit_future),
        sys.executable, '-c', code,
        stdin=None, stderr=None)

    # Wait for the subprocess exit using the process_exited()
    # method of the protocol.
    await exit_future

    # Close the stdout pipe.
    transport.close()

    # Read the output which was collected by the
```

(suite sur la page suivante)

(suite de la page précédente)

```
    # pipe_data_received() method of the protocol.
    data = bytes(protocol.output)
    return data.decode('ascii').rstrip()

date = asyncio.run(get_date())
print(f"Current date: {date}")
```

Voir aussi le *même exemple* écrit à l'aide d'API de haut niveau.

19.1.12 Politiques

Avertissement

Policies are deprecated and will be removed in Python 3.16. Users are encouraged to use the `asyncio.run()` function or the `asyncio.Runner` with *loop_factory* to use the desired loop implementation.

Une politique de boucle d'événements est un objet global utilisé pour obtenir et définir la *boucle d'événements* actuelle, ainsi que créer de nouvelles boucles d'événements. La politique par défaut peut être *remplacée* par des *alternatives intégrées* afin d'utiliser d'autres implémentations de boucles d'événements, ou remplacée par une *politique personnalisée* qui peut remplacer ces comportements.

Un *objet politique* définit la notion de *contexte* et gère une boucle d'événement distincte par contexte. Ceci est valable fil par fil d'exécution par défaut, bien que les politiques personnalisées puissent définir le *contexte* différemment.

En utilisant une politique de boucle d'événement personnalisée, le comportement des fonctions `get_event_loop()`, `set_event_loop()` et `new_event_loop()` peut être personnalisé.

Les objets politiques doivent implémenter les API définies dans la classe mère abstraite `AbstractEventLoopPolicy`.

Obtenir et définir la politique

Les fonctions suivantes peuvent être utilisées pour obtenir et définir la politique du processus en cours :

`asyncio.`**`get_event_loop_policy`**`()`

Renvoie la politique actuelle à l'échelle du processus.

Obsolète depuis la version 3.14 : The `get_event_loop_policy()` function is deprecated and will be removed in Python 3.16.

`asyncio.`**`set_event_loop_policy`**`(`*policy*`)`

Définit la politique actuelle sur l'ensemble du processus sur *policy*.

Si *policy* est définie sur `None`, la politique par défaut est restaurée.

Obsolète depuis la version 3.14 : The `set_event_loop_policy()` function is deprecated and will be removed in Python 3.16.

Objets politiques

La classe mère abstraite de la politique de boucle d'événements est définie comme suit :

`class` `asyncio.`**`AbstractEventLoopPolicy`**

Une classe de base abstraite pour les politiques *asyncio*.

`get_event_loop``()`

Récupère la boucle d'évènements pour le contexte actuel.

Renvoie un objet de boucle d'événements en implémentant l'interface `AbstractEventLoop`.

Cette méthode ne devrait jamais renvoyer `None`.

Modifié dans la version 3.6.

set_event_loop(*loop*)

Définit la boucle d'événements du contexte actuel sur *loop*.

new_event_loop()

Crée et renvoie un nouvel objet de boucle d'événements.

Cette méthode ne devrait jamais renvoyer `None`.

Obsolète depuis la version 3.14 : The `AbstractEventLoopPolicy` class is deprecated and will be removed in Python 3.16.

asyncio est livré avec les politiques intégrées suivantes :

class asyncio.**DefaultEventLoopPolicy**

La politique *asyncio* par défaut. Utilise `SelectorEventLoop` sur les plates-formes Unix et `ProactorEventLoop` sur Windows.

Il n'est pas nécessaire d'installer la politique par défaut manuellement. *asyncio* est configuré pour utiliser automatiquement la politique par défaut.

Modifié dans la version 3.8 : Sous Windows, `ProactorEventLoop` est désormais utilisée par défaut.

Modifié dans la version 3.14 : The `get_event_loop()` method of the default asyncio policy now raises a `RuntimeError` if there is no set event loop.

Obsolète depuis la version 3.14 : The `DefaultEventLoopPolicy` class is deprecated and will be removed in Python 3.16.

class asyncio.**WindowsSelectorEventLoopPolicy**

Politique de boucle d'événements alternative utilisant l'implémentation de la boucle d'événements `ProactorEventLoop`.

Availability : Windows.

Obsolète depuis la version 3.14 : The `WindowsSelectorEventLoopPolicy` class is deprecated and will be removed in Python 3.16.

class asyncio.**WindowsProactorEventLoopPolicy**

Politique de boucle d'événements alternative utilisant l'implémentation de la boucle d'événements `ProactorEventLoop`.

Availability : Windows.

Obsolète depuis la version 3.14 : The `WindowsProactorEventLoopPolicy` class is deprecated and will be removed in Python 3.16.

Politiques personnalisées

Pour implémenter une nouvelle politique de boucle d'événements, il est recommandé de sous-classer `DefaultEventLoopPolicy` et de réimplémenter les méthodes pour lesquelles un comportement personnalisé est souhaité, par exemple :

```
class MyEventLoopPolicy(asyncio.DefaultEventLoopPolicy):

    def get_event_loop(self):
        """Get the event loop.

        This may be None or an instance of EventLoop.
        """
        loop = super().get_event_loop()
        # Do something with loop ...
        return loop
```

(suite sur la page suivante)

(suite de la page précédente)

```
asyncio.set_event_loop_policy(MyEventLoopPolicy())
```

19.1.13 Prise en charge de la plate-forme

Le module `asyncio` est conçu pour être portable, mais certaines plates-formes présentent des différences et des limitations subtiles en raison de l'architecture et des capacités sous-jacentes des plates-formes.

Toutes plateformes

— `loop.add_reader()` et `loop.add_writer()` ne peuvent pas être utilisées pour surveiller les entrées-sorties de fichiers.

Windows

Code source : Lib/asyncio/proactor_events.py, Lib/asyncio/windows_events.py, Lib/asyncio/windows_utils.py

---

Modifié dans la version 3.8 : sous Windows, `ProactorEventLoop` est désormais la boucle d'événements par défaut.

Aucune boucle d'événements sous Windows ne prend en charge les méthodes suivantes :

— `loop.create_unix_connection()` and `loop.create_unix_server()` are not supported. The `socket.AF_UNIX` socket family is specific to Unix.

— `loop.add_signal_handler()` et `loop.remove_signal_handler()` ne sont pas prises en charge.

`SelectorEventLoop` a les limitations suivantes :

— `SelectSelector` est utilisée pour attendre les événements de connecteur (*sockets*) : elle prend en charge les connecteurs et est limitée à 512 connecteurs.

— `loop.add_reader()` et `loop.add_writer()` n'acceptent que les descripteurs de connecteur (par exemple, les descripteurs de tube fichier ne sont pas pris en charge).

— Les tubes ne sont pas pris en charge, donc les méthodes `loop.connect_read_pipe()` et `loop.connect_write_pipe()` ne sont pas implémentées.

— Les *sous-processus* ne sont pas pris en charge, c'est-à-dire que les méthodes `loop.subprocess_exec()` et `loop.subprocess_shell()` ne sont pas implémentées.

`ProactorEventLoop` a les limitations suivantes :

— Les méthodes `loop.add_reader()` et `loop.add_writer()` ne sont pas prises en charge.

La résolution de l'horloge monotone sous Windows est généralement d'environ 15,6 millisecondes. La meilleure résolution est de 0,5 milliseconde. La résolution dépend du matériel (disponibilité de HPET) et de la configuration de Windows.

Prise en charge des sous-processus sous Windows

Sous Windows, la boucle d'événements par défaut `ProactorEventLoop` prend en charge les sous-processus, contrairement à `SelectorEventLoop`.

macOS

Les versions modernes de macOS sont entièrement prises en charge.

macOS ≤ 10.8

Sur macOS 10.6, 10.7 et 10.8, la boucle d'événements par défaut utilise `selectors.KqueueSelector`, qui ne prend pas en charge les périphériques de caractères sur ces versions. La `SelectorEventLoop` peut être configurée manuellement pour utiliser `SelectSelector` ou `PollSelector` pour prendre en charge les périphériques de caractères sur ces anciennes versions de macOS. Par exemple :

```
import asyncio
import selectors

selector = selectors.SelectSelector()
loop = asyncio.SelectorEventLoop(selector)
asyncio.set_event_loop(loop)
```

19.1.14 Extension

La direction principale pour l'extension d'`asyncio` est l'écriture de classes *event loop* personnalisées. *Asyncio* a des assistants qui peuvent être utilisés pour simplifier cette tâche.

> **Note**
>
> la réutilisation du code asynchrone existant doit se faire avec prudence, une nouvelle version de Python est libre de rompre la compatibilité descendante dans la partie *interne* de l'API.

Écriture d'une boucle d'événements personnalisée

`asyncio.AbstractEventLoop` déclare de très nombreuses méthodes. Les mettre en œuvre à partir de zéro est un travail fastidieux.

Une boucle d'événements peut obtenir gratuitement l'implémentation de nombreuses méthodes courantes en héritant de `asyncio.BaseEventLoop`.

Le successeur doit, pour ce qui le concerne, implémenter un ensemble de méthodes *privées* déclarées mais non implémentées dans `asyncio.BaseEventLoop`.

Par exemple, `loop.create_connection()` vérifie les arguments, résout les adresses DNS et appelle `loop._make_socket_transport()` qui doit être implémentée par la classe héritée. La méthode `_make_socket_transport()` n'est pas documentée et est considérée comme une API *interne*.

Constructeurs privés *Future* et *Task*

`asyncio.Future` et `asyncio.Task` ne doivent jamais être créées directement, veuillez utiliser les correspondances `loop.create_future()` et `loop.create_task()`, ou les fabriques `asyncio.create_task()` à la place.

Cependant, les *boucles d'événements* tierces peuvent *réutiliser* les implémentations de *Future* et de *Task* intégrées dans le but d'obtenir gratuitement un code complexe et hautement optimisé.

À cette fin, les constructeurs *privés* suivants sont répertoriés :

`Future.__init__`(*, *loop=None*)

Crée une instance de la *future* native.

loop est une instance de boucle d'événements facultative.

`Task.__init__`(*coro*, *, *loop=None*, *name=None*, *context=None*)

Crée une instance de la *Task* native.

loop est une instance de boucle d'événements facultative. Le reste des arguments est décrit dans la description de `loop.create_task()`.

Modifié dans la version 3.11 : l'argument *contexte* a été ajouté.

Prise en charge de la durée de vie des tâches

A third party task implementation should call the following functions to keep a task visible by `asyncio.all_tasks()` and `asyncio.current_task()` :

`asyncio.`**`_register_task`**(*task*)

Enregistre une nouvelle *tâche* comme gérée par *asyncio*.

Appelez la fonction à partir d'un constructeur de tâche.

`asyncio.`**`_unregister_task`**(*task*)

Désinscrit une *tâche* des structures internes *asyncio*.

La fonction doit être appelée lorsqu'une tâche est sur le point de se terminer.

`asyncio.`**`_enter_task`**(*loop*, *task*)

Bascule la tâche en cours vers l'argument *task*.

Appelez la fonction juste avant d'exécuter une partie de la *coroutine* intégrée (`coroutine.send()` ou `coroutine.throw()`).

`asyncio.`**`_leave_task`**(*loop*, *task*)

Rebascule la tâche en cours de *task* à `None`.

Appelez la fonction juste après l'exécution de `coroutine.send()` ou `coroutine.throw()`.

19.1.15 Index de l'API de haut niveau

Cette page répertorie toutes les API *async/await* de haut niveau disponibles dans l'API *asyncio*.

Tâches

Utilitaires pour exécuter des programmes asynchrones, créer des tâches et attendre plusieurs choses avec des délais maximaux d'attente.

`run()`	Crée une boucle d'événements, exécute une coroutine, ferme la boucle.
`Runner`	Gestionnaire de contexte qui simplifie plusieurs appels de fonction asynchrones.
`Task`	Objet de tâche.
`TaskGroup`	Gestionnaire de contexte pour un groupe de tâches. Fournit un moyen pratique et fiable d'attendre la fin de toutes les tâches du groupe.
`create_task()`	Démarre une tâche asynchrone, puis la renvoie.
`current_task()`	Renvoie la tâche actuelle.
`all_tasks()`	Renvoie toutes les tâches qui ne sont pas encore terminées pour une boucle d'événements.
`await sleep()`	Dort quelques secondes.
`await gather()`	Planifie et attend des choses concurrentes.
`await wait_for()`	Exécute avec un délai d'attente.
`await shield()`	Empêche l'annulation.
`await wait()`	Surveille l'achèvement.
`timeout()`	Exécute avec un délai d'attente. Utile dans les cas où `wait_for` ne convient pas.
`to_thread()`	Exécute de manière asynchrone une fonction dans un fil d'exécution séparé.
`run_coroutine_threadsafe()`	Planifie une coroutine à partir d'un autre fil d'exécution.
`for in as_completed()`	Surveille l'achèvement avec une boucle `for`.

Exemples

- *Utilisation d'asyncio.gather() pour exécuter des choses en parallèle*.
- *Utilisation d'asyncio.wait_for() pour appliquer un délai d'attente*.
- *Annulation*.
- *Utilisation d'asyncio.sleep()*.
- Voir aussi la page de *documentation de Tasks*.

Files d'attente

Les files d'attente doivent être utilisées pour répartir le travail entre plusieurs tâches asynchrones, implémenter des pools de connexions et des modèles *pub/sub*.

`Queue`	File d'attente FIFO (premier entré, premier sorti).
`PriorityQueue`	File d'attente avec gestion des priorités.
`LifoQueue`	Pile LIFO (dernier entré, premier sorti).

Exemples

- *Utilisation d'asyncio.Queue pour répartir la charge de travail entre plusieurs tâches*.
- Voir aussi la page de *documentation des Queues*.

Sous-processus

Utilitaires pour générer des sous-processus et exécuter des commandes shell.

`await create_subprocess_exec()`	Crée un sous-processus.
`await create_subprocess_shell()`	Exécute une commande shell.

Exemples

- *Exécution d'une commande shell*.
- Voir aussi la documentation des *API subprocess*.

Flux (*streams*)

API de haut niveau pour travailler avec les entrées-sorties réseau.

`await open_connection()`	Établit une connexion TCP.
`await open_unix_connection()`	Établit une connexion sur un connecteur Unix.
`await start_server()`	Démarre un serveur TCP.
`await start_unix_server()`	Démarre un serveur de socket Unix.
`StreamReader`	Objet *async/await* de haut niveau pour recevoir des données réseau.
`StreamWriter`	Objet *async/await* de haut niveau pour envoyer des données réseau.

Exemples

- *Exemple de client TCP*.
- Voir aussi la documentation des *API streams*.

Synchronisation

Primitives de synchronisation de type thread qui peuvent être utilisées dans les tâches.

`Lock`	Verrou mutex.
`Event`	Objet événement.
`Condition`	Objet condition.
`Semaphore`	Sémaphore.
`BoundedSemaphore`	Sémaphore avec des bornes.
`Barrier`	Objet barrière.

Exemples

- *Utilisation d'asyncio.Event*.
- *Utilisation d'asyncio.Barrier*.
- Voir aussi la documentation des *primitives asyncio de synchronisation*.

Exceptions

`asyncio.CancelledError`	Levée lorsqu'une tâche est annulée. Voir aussi `Task.cancel()`.
`asyncio.BrokenBarrierError`	Levée lorsqu'une barrière est brisée. Voir aussi `Barrier.wait()`.

Exemples

- *Utilisation de CancelledError pour exécuter le code sur la demande d'annulation*.
- Voir aussi la liste complète des *exceptions spécifiques asyncio*.

19.1.16 Index de l'API de bas niveau

Cette page répertorie toutes les API asynchrones de bas niveau.

Obtenir une boucle d'évènements

`asyncio.get_running_loop()`	La fonction **préférée** pour obtenir la boucle d'événements en cours d'exécution.
`asyncio.get_event_loop()`	Renvoie une instance de boucle d'événements (en cours d'exécution ou actuelle via la politique actuelle).
`asyncio.set_event_loop()`	Définit la boucle d'événements comme actuelle via la politique actuelle.
`asyncio.new_event_loop()`	Crée une nouvelle boucle d'événements.

Exemples

- *Utilisation d'asyncio.get_running_loop()*.

Méthodes de la boucle d'évènements

Voir aussi la section principale de la documentation sur les *Méthodes de la boucle d'évènements*.

Cycle de vie

`loop.run_until_complete()`	Exécute un Future/Task/awaitable jusqu'à ce qu'il soit terminé.
`loop.run_forever()`	Exécute la boucle d'événements pour toujours.
`loop.stop()`	Arrête l'exécution de la boucle d'évènements.
`loop.close()`	Arrête la boucle d'évènements.
`loop.is_running()`	Renvoie `True` si la boucle d'événements est en cours d'exécution.
`loop.is_closed()`	Renvoie `True` si la boucle d'évènements est arrêtée.
`await loop.shutdown_asyncgens()`	Ferme les générateurs asynchrones.

Débogage

`loop.set_debug()`	Active ou désactive le mode débogage.
`loop.get_debug()`	Renvoie le mode de débogage actuel.

Planification des rappels

`loop.call_soon()`	Invoque un rappel bientôt.
`loop.call_soon_threadsafe()`	Une variante compatible avec les programmes à fils d'exécution multiples de `loop.call_soon()`.
`loop.call_later()`	Invoque un rappel *après* le temps imparti.
`loop.call_at()`	Invoque un rappel *à l'heure* indiquée.

Thread/Interpreter/Process Pool

`await loop.run_in_executor()`	Exécute une fonction utilisant beaucoup de CPU ou une autre fonction bloquante dans un exécuteur `concurrent.futures`.
`loop.set_default_executor()`	Définit l'exécuteur par défaut pour `loop.run_in_executor()`.

Tâches et *Futures*

`loop.create_future()`	Crée un objet `Future`.
`loop.create_task()`	Planifie la coroutine en tant que `Task`.
`loop.set_task_factory()`	Définit une fabrique utilisée par `loop.create_task()` pour créer des `Tasks`.
`loop.get_task_factory()`	Récupère la fabrique que `loop.create_task()` utilise pour créer des `Tasks`.

DNS

`await loop.getaddrinfo()`	Version asynchrone de `socket.getaddrinfo()`.
`await loop.getnameinfo()`	Version asynchrone de `socket.getnameinfo()`.

Réseau et communication inter-processus

`await loop.create_connection()`	Ouvre une connexion TCP.
`await loop.create_server()`	Crée un serveur TCP.
`await loop.create_unix_connection()`	Ouvre une connexion socket Unix.
`await loop.create_unix_server()`	Crée un serveur de socket Unix.
`await loop.connect_accepted_socket()`	Enveloppe une `socket` dans une paire `(transport, protocol)`.
`await loop.create_datagram_endpoint()`	Ouvre une connexion par datagramme (UDP).
`await loop.sendfile()`	Envoie un fichier via un transport.
`await loop.start_tls()`	Bascule une connexion existante vers TLS.
`await loop.connect_read_pipe()`	Enveloppe une extrémité de lecture d'un tube dans une paire `(transport, protocol)`.
`await loop.connect_write_pipe()`	Enveloppe une extrémité d'écriture d'un tube dans une paire `(transport, protocol)`.

Interfaces de connexion (*sockets*)

`await loop.sock_recv()`	Reçoit les données de `socket`.
`await loop.sock_recv_into()`	Reçoit les données de `socket` dans un tampon.
`await loop.sock_recvfrom()`	Reçoit un datagramme de `socket`.
`await loop.sock_recvfrom_into()`	Reçoit un datagramme de `socket` dans un tampon.
`await loop.sock_sendall()`	Envoie des données à `socket`.
`await loop.sock_sendto()`	Envoie un datagramme via `socket` à l'adresse indiquée.
`await loop.sock_connect()`	Connecte la `socket`.
`await loop.sock_accept()`	Accepte une connexion `socket`.
`await loop.sock_sendfile()`	Envoie un fichier via `socket`.
`loop.add_reader()`	Commence à observer un descripteur de fichier pour la disponibilité en lecture.
`loop.remove_reader()`	Arrête d'observer un descripteur de fichier pour la disponibilité en lecture.
`loop.add_writer()`	Commence à observer un descripteur de fichier pour la disponibilité en écriture.
`loop.remove_writer()`	Arrête d'observer un descripteur de fichier pour la disponibilité en écriture.

Signaux Unix

`loop.add_signal_handler()`	Ajoute un gestionnaire pour le `signal`.
`loop.remove_signal_handler()`	Supprime un gestionnaire pour le `signal`.

Sous-processus

`loop.subprocess_exec()`	Crée un sous-processus.
`loop.subprocess_shell()`	Crée un sous-processus à partir d'une commande shell.

Gestion des erreurs

`loop.call_exception_handler()`	Appelle le gestionnaire d'exceptions.
`loop.set_exception_handler()`	Définit un nouveau gestionnaire d'exceptions.
`loop.get_exception_handler()`	Renvoie le gestionnaire d'exceptions actuel.
`loop.default_exception_handler()`	Implémentation du gestionnaire d'exceptions par défaut.

Exemples

- *Utilisation d'asyncio.new_event_loop() et de loop.run_forever()*.
- *Utilisation de loop.call_later()*.
- Utilisation de `loop.create_connection()` pour implémenter *un client écho*.
- Utilisation de `loop.create_connection()` pour *connecter un socket*.
- *Utilisation de add_reader() pour surveiller un descripteur de fichier pour les événements de lecture*.
- *Utilisation de loop.add_signal_handler()*.
- *Utilisation de loop.subprocess_exec()*.

Transports

Tous les transports mettent en œuvre les méthodes suivantes :

`transport.close()`	Ferme le transport.
`transport.is_closing()`	Renvoie `True` si le transport se ferme ou est fermé.
`transport.get_extra_info()`	Demande d'informations sur le transport.
`transport.set_protocol()`	Change le protocole.
`transport.get_protocol()`	Renvoie le protocole courant.

Transports pouvant recevoir des données (connexions TCP et Unix, pipes, etc.). Renvoyé par des méthodes telles que `loop.create_connection()`, `loop.create_unix_connection()`, `loop.connect_read_pipe()`, etc :

Transports en lecture

`transport.is_reading()`	Renvoie `True` si le transport est en cours de réception.
`transport.pause_reading()`	Suspend la réception.
`transport.resume_reading()`	Reprend la réception.

Transports pouvant envoyer des données (connexions TCP et Unix, pipes, etc.). Renvoyé par des méthodes telles que `loop.create_connection()`, `loop.create_unix_connection()`, `loop.connect_write_pipe()`, etc :

Transports en écriture

`transport.write()`	Écrit des données dans le transport.
`transport.writelines()`	Écrit des tampons dans le transport.
`transport.can_write_eof()`	Renvoie `True` si le transport prend en charge l'envoi d'EOF.
`transport.write_eof()`	Ferme et envoie EOF après avoir vidé les données mises en mémoire tampon.
`transport.abort()`	Ferme immédiatement le transport.
`transport.get_write_buffer_size()`	Return the current size of the output buffer.
`transport.get_write_buffer_limits()`	Renvoie les seuils haut et bas pour le contrôle du flux d'écriture.
`transport.set_write_buffer_limits()`	Définit de nouvelles bornes hautes et basses pour le contrôle du flux d'écriture.

Transports renvoyés par `loop.create_datagram_endpoint()` :

Transports par datagrammes

`transport.sendto()`	Envoie des données au pair distant.
`transport.abort()`	Ferme immédiatement le transport.

Abstraction de transport de bas niveau sur les sous-processus. Renvoyé par `loop.subprocess_exec()` et `loop.subprocess_shell()` :

Transports vers des sous-processus

`transport.get_pid()`	Renvoie l'ID de processus du sous-processus.
`transport.get_pipe_transport()`	Renvoie le transport pour le tube de communication demandé (*stdin*, *stdout* ou *stderr*).
`transport.get_returncode()`	Renvoie le code de retour du sous-processus.
`transport.kill()`	Tue le sous-processus.
`transport.send_signal()`	Envoie un signal au sous-processus.
`transport.terminate()`	Termine le sous-processus.
`transport.close()`	Tue le sous-processus et ferme tous les tubes.

Protocoles

Les classes de protocole peuvent implémenter les **méthodes de rappel** suivantes :

`callback` `connection_made()`	Appelée lorsqu'une connexion est établie.
`callback` `connection_lost()`	Appelé lorsqu'une connexion est perdue ou fermée.
`callback` `pause_writing()`	Appelée lorsque la mémoire tampon du transport dépasse le seuil haut.
`callback` `resume_writing()`	Appelée lorsque la mémoire tampon du transport passe sous le seuil bas.

Protocoles de flux (TCP, Unix Sockets, Pipes)

`callback` `data_received()`	Appelée lorsque certaines données sont reçues.
`callback` `eof_received()`	Appelée lorsqu'un EOF est reçu.

Protocoles de flux tamponnés

`callback` `get_buffer()`	Appelée pour allouer un nouveau tampon de réception.
`callback` `buffer_updated()`	Appelée lorsque le tampon a été mis à jour avec les données reçues.
`callback` `eof_received()`	Appelée lorsqu'un EOF est reçu.

Protocoles par datagrammes (non-connectés)

`callback` `datagram_received()`	Appelée lorsqu'un datagramme est reçu.
`callback` `error_received()`	Appelée lorsqu'une opération d'envoi ou de réception précédente lève une `OSError`.

Protocoles liés aux sous-processus

callback *pipe_data_received()*	Appelée lorsqu'un processus enfant écrit sur sa sortie d'erreur ou sa sortie standard.
callback *pipe_connection_lost()*	Appelée lorsqu'un tube de communication avec un sous-processus est fermée.
callback *process_exited()*	Called when the child process has exited. It can be called before *pipe_data_received()* and *pipe_connection_lost()* methods.

Politiques de boucle d'événements

Les politiques sont un mécanisme de bas niveau pour modifier le comportement de fonctions telles que *asyncio.get_event_loop()*. Voir aussi la section principale *Politiques* pour plus de détails.

Accès aux politiques

asyncio.get_event_loop_policy()	Renvoie la politique actuelle à l'échelle du processus.
asyncio.set_event_loop_policy()	Définit une nouvelle politique à l'échelle du processus.
AbstractEventLoopPolicy	Classe mère pour les objets de politique.

19.1.17 Programmer avec *asyncio*

La programmation asynchrone est différente de la programmation « séquentielle » classique.

Cette page liste les pièges et erreurs communs que le développeur pourrait rencontrer et décrit comment les éviter.

Mode débogage

Par défaut, *asyncio* s'exécute en mode production. Pour faciliter le développement, *asyncio* possède un « mode débogage ».

Il existe plusieurs façons d'activer le mode débogage de *asyncio* :
- en réglant la variable d'environnement `PYTHONASYNCIODEBUG` à `1` ;
- en utilisant le mode développement de Python (*Python Development Mode*) ;
- en passant `debug=True` à la fonction *asyncio.run()* ;
- en appelant la méthode *loop.set_debug()*.

En plus d'activer le mode débogage, vous pouvez également :
- setting the log level of the *asyncio logger* to *logging.DEBUG*, for example the following snippet of code can be run at startup of the application :

```
logging.basicConfig(level=logging.DEBUG)
```

- configurer le module *warnings* afin d'afficher les avertissements de type *ResourceWarning* ; vous pouvez faire cela en utilisant l'option `-W default` sur la ligne de commande.

Lorsque le mode débogage est activé :
- *asyncio* surveille les *coroutines qui ne sont jamais attendues* et les journalise ; cela atténue le problème des « *await* oubliés » ;
- beaucoup d'*API asyncio* ne prenant pas en charge les fils d'exécution multiples (comme les méthodes *loop.call_soon()* et *loop.call_at()*) lèvent une exception si elles sont appelées par le mauvais fil d'exécution ;
- le temps d'exécution du sélecteur d'entrée-sortie est journalisé si une opération prend trop de temps à s'effectuer ;
- les fonctions de rappel prenant plus de 100 ms sont journalisées ; l'attribut *loop.slow_callback_duration* peut être utilisé pour changer la limite (en secondes) après laquelle une fonction de rappel est considérée comme « lente ».

Programmation concurrente et multi-fils

Une boucle d'évènements s'exécute dans un fil d'exécution (typiquement dans le fil principal) et traite toutes les fonctions de rappel (*callbacks*) ainsi que toutes les tâches dans ce même fil. Lorsqu'une tâche est en cours d'exécution dans la boucle d'évènements, aucune autre tâche ne peut s'exécuter dans ce fil. Quand une tâche traite une expression `await`, elle se suspend et laisse la boucle d'évènements traiter la tâche suivante.

Pour planifier un *rappel* depuis un autre fil d'exécution système, utilisez la méthode `loop.call_soon_threadsafe()`. Par exemple :

```
loop.call_soon_threadsafe(callback, *args)
```

La plupart des objets *asyncio* ne sont pas conçus pour être exécutés dans un contexte multi-fils (*thread-safe*) mais cela n'est en général pas un problème à moins que l'objet ne fasse appel à du code se trouvant en dehors d'une tâche ou d'une fonction de rappel. Dans ce dernier cas, si le code appelle les *API* bas niveau de *asyncio*, utilisez la méthode `loop.call_soon_threadsafe()`. Par exemple :

```
loop.call_soon_threadsafe(fut.cancel)
```

Pour planifier un objet concurrent depuis un autre fil d'exécution système, utilisez `run_coroutine_threadsafe()`. Cette fonction renvoie un objet `concurrent.futures.Future` pour accéder au résultat :

```
async def coro_func():
    return await asyncio.sleep(1, 42)

# Later in another OS thread:

future = asyncio.run_coroutine_threadsafe(coro_func(), loop)
# Wait for the result:
result = future.result()
```

To handle signals the event loop must be run in the main thread.

The `loop.run_in_executor()` method can be used with a `concurrent.futures.ThreadPoolExecutor` or `InterpreterPoolExecutor` to execute blocking code in a different OS thread without blocking the OS thread that the event loop runs in.

Il n'y a actuellement aucune façon de planifier des coroutines ou des rappels directement depuis un autre processus (comme, par exemple, un processus démarré avec `multiprocessing`). La section *Méthodes de la boucle d'évènements* liste les *API* pouvant lire les tubes (*pipes*) et surveiller les descripteurs de fichiers sans bloquer la boucle d'évènements. De plus, les *API* *Subprocess* d'*asyncio* fournissent un moyen de démarrer un processus et de communiquer avec lui depuis la boucle d'évènements. Enfin, la méthode `loop.run_in_executor()` susmentionnée peut également être utilisée avec `concurrent.futures.ProcessPoolExecutor` pour exécuter du code dans un processus différent.

Exécution de code bloquant

Du code bloquant sur des opérations de calcul (*CPU-bound*) ne devrait pas être appelé directement. Par exemple, si une fonction effectue des calculs utilisant le CPU intensivement pendant une seconde, toutes les tâches *asyncio* concurrentes et les opérations d'entrées-sorties seront bloquées pour une seconde.

An executor can be used to run a task in a different thread, including in a different interpreter, or even in a different process to avoid blocking the OS thread with the event loop. See the `loop.run_in_executor()` method for more details.

Journalisation

Asyncio utilise le module `logging`. Toutes les opérations de journalisation sont effectuées via l'enregistreur (*logger*) `"asyncio"`.

The default log level is `logging.INFO`, which can be easily adjusted :

```
logging.getLogger("asyncio").setLevel(logging.WARNING)
```

La journalisation réseau peut bloquer la boucle d'événements. Il est recommandé d'utiliser un fil d'exécution séparé pour gérer les journaux ou d'utiliser des entrées-sorties non bloquantes. Par exemple, voir blocking-handlers.

Détection des coroutines jamais attendues

Lorsqu'une fonction coroutine est appelée mais qu'elle n'est pas attendue (p. ex. `coro()` au lieu de `await coro()`) ou si la coroutine n'est pas planifiée avec `asyncio.create_task()`, *asyncio* émet un `RuntimeWarning` :

```
import asyncio

async def test():
    print("never scheduled")

async def main():
    test()

asyncio.run(main())
```

Sortie :

```
test.py:7: RuntimeWarning: coroutine 'test' was never awaited
  test()
```

Affichage en mode débogage :

```
test.py:7: RuntimeWarning: coroutine 'test' was never awaited
Coroutine created at (most recent call last)
  File "../t.py", line 9, in <module>
    asyncio.run(main(), debug=True)

  < .. >

  File "../t.py", line 7, in main
    test()
  test()
```

La façon habituelle de régler ce problème est d'attendre (*await*) la coroutine ou bien d'appeler la fonction `asyncio.create_task()` :

```
async def main():
    await test()
```

Détection des exceptions jamais récupérées

Si la méthode `Future.set_exception()` est appelée mais que l'objet *Future* n'est pas attendu, l'exception n'est pas propagée au code utilisateur. Dans ce cas, *asyncio* écrit un message dans le journal lorsque l'objet *Future* est récupéré par le ramasse-miette.

Exemple d'une exception non-gérée :

```
import asyncio

async def bug():
    raise Exception("not consumed")

async def main():
```

(suite sur la page suivante)

(suite de la page précédente)

```
    asyncio.create_task(bug())

asyncio.run(main())
```

Sortie :

```
Task exception was never retrieved
future: <Task finished coro=<bug() done, defined at test.py:3>
  exception=Exception('not consumed')>

Traceback (most recent call last):
  File "test.py", line 4, in bug
    raise Exception("not consumed")
Exception: not consumed
```

Activez le mode débogage pour récupérer la trace d'appels indiquant où la tâche a été créée :

```
asyncio.run(main(), debug=True)
```

Affichage en mode débogage :

```
Task exception was never retrieved
future: <Task finished coro=<bug() done, defined at test.py:3>
    exception=Exception('not consumed') created at asyncio/tasks.py:321>

source_traceback: Object created at (most recent call last):
  File "../t.py", line 9, in <module>
    asyncio.run(main(), debug=True)

< .. >

Traceback (most recent call last):
  File "../t.py", line 4, in bug
    raise Exception("not consumed")
Exception: not consumed
```

Note

Le code source d'`asyncio` est disponible dans Lib/asyncio/.

19.2 `socket` --- Low-level networking interface

Code source : Lib/secrets.py

This module provides access to the BSD *socket* interface. It is available on all modern Unix systems, Windows, MacOS, and probably additional platforms.

Note

Some behavior may be platform dependent, since calls are made to the operating system socket APIs.

Availability : not WASI.

This module does not work or is not available on WebAssembly. See *Plateformes WebAssembly* for more information.

The Python interface is a straightforward transliteration of the Unix system call and library interface for sockets to Python's object-oriented style : the `socket()` function returns a *socket object* whose methods implement the various socket system calls. Parameter types are somewhat higher-level than in the C interface : as with `read()` and `write()` operations on Python files, buffer allocation on receive operations is automatic, and buffer length is implicit on send operations.

Voir aussi

Module `socketserver`
Classes that simplify writing network servers.

Module `ssl`
A TLS/SSL wrapper for socket objects.

19.2.1 Socket families

Depending on the system and the build options, various socket families are supported by this module.

The address format required by a particular socket object is automatically selected based on the address family specified when the socket object was created. Socket addresses are represented as follows :

- The address of an `AF_UNIX` socket bound to a file system node is represented as a string, using the file system encoding and the `'surrogateescape'` error handler (see **PEP 383**). An address in Linux's abstract namespace is returned as a *bytes-like object* with an initial null byte ; note that sockets in this namespace can communicate with normal file system sockets, so programs intended to run on Linux may need to deal with both types of address. A string or bytes-like object can be used for either type of address when passing it as an argument.

  Modifié dans la version 3.3 : Previously, `AF_UNIX` socket paths were assumed to use UTF-8 encoding.

  Modifié dans la version 3.5 : N'importe quel *bytes-like object* est maintenant accepté.
- A pair `(host, port)` is used for the `AF_INET` address family, where *host* is a string representing either a hostname in internet domain notation like `'daring.cwi.nl'` or an IPv4 address like `'100.50.200.5'`, and *port* is an integer.
  - For IPv4 addresses, two special forms are accepted instead of a host address : `''` represents `INADDR_ANY`, which is used to bind to all interfaces, and the string `'<broadcast>'` represents `INADDR_BROADCAST`. This behavior is not compatible with IPv6, therefore, you may want to avoid these if you intend to support IPv6 with your Python programs.
- For `AF_INET6` address family, a four-tuple `(host, port, flowinfo, scope_id)` is used, where *flowinfo* and *scope_id* represent the `sin6_flowinfo` and `sin6_scope_id` members in `struct sockaddr_in6` in C. For `socket` module methods, *flowinfo* and *scope_id* can be omitted just for backward compatibility. Note, however, omission of *scope_id* can cause problems in manipulating scoped IPv6 addresses.

  Modifié dans la version 3.7 : For multicast addresses (with *scope_id* meaningful) *address* may not contain `%scope_id` (or `zone id`) part. This information is superfluous and may be safely omitted (recommended).
- `AF_NETLINK` sockets are represented as pairs `(pid, groups)`.
- Linux-only support for TIPC is available using the `AF_TIPC` address family. TIPC is an open, non-IP based networked protocol designed for use in clustered computer environments. Addresses are represented by a tuple, and the fields depend on the address type. The general tuple form is `(addr_type, v1, v2, v3 [, scope])`, where :
  - *addr_type* is one of `TIPC_ADDR_NAMESEQ`, `TIPC_ADDR_NAME`, or `TIPC_ADDR_ID`.
  - *scope* is one of `TIPC_ZONE_SCOPE`, `TIPC_CLUSTER_SCOPE`, and `TIPC_NODE_SCOPE`.
  - If *addr_type* is `TIPC_ADDR_NAME`, then *v1* is the server type, *v2* is the port identifier, and *v3* should be 0.

    If *addr_type* is `TIPC_ADDR_NAMESEQ`, then *v1* is the server type, *v2* is the lower port number, and *v3* is the upper port number.

    If *addr_type* is `TIPC_ADDR_ID`, then *v1* is the node, *v2* is the reference, and *v3* should be set to 0.
- A tuple `(interface, )` is used for the `AF_CAN` address family, where *interface* is a string representing a

network interface name like `'can0'`. The network interface name `''` can be used to receive packets from all network interfaces of this family.
  - *CAN_ISOTP* protocol require a tuple `(interface, rx_addr, tx_addr)` where both additional parameters are unsigned long integer that represent a CAN identifier (standard or extended).
  - *CAN_J1939* protocol require a tuple `(interface, name, pgn, addr)` where additional parameters are 64-bit unsigned integer representing the ECU name, a 32-bit unsigned integer representing the Parameter Group Number (PGN), and an 8-bit integer representing the address.
- A string or a tuple `(id, unit)` is used for the `SYSPROTO_CONTROL` protocol of the `PF_SYSTEM` family. The string is the name of a kernel control using a dynamically assigned ID. The tuple can be used if ID and unit number of the kernel control are known or if a registered ID is used.

  Ajouté dans la version 3.3.
- `AF_BLUETOOTH` supports the following protocols and address formats :
  - `BTPROTO_L2CAP` accepts `(bdaddr, psm)` where `bdaddr` is the Bluetooth address as a string and `psm` is an integer.
  - `BTPROTO_RFCOMM` accepts `(bdaddr, channel)` where `bdaddr` is the Bluetooth address as a string and `channel` is an integer.
  - `BTPROTO_HCI` accepts `(device_id,)` where `device_id` is either an integer or a string with the Bluetooth address of the interface. (This depends on your OS ; NetBSD and DragonFlyBSD expect a Bluetooth address while everything else expects an integer.)

    Modifié dans la version 3.2 : NetBSD and DragonFlyBSD support added.
  - `BTPROTO_SCO` accepts `bdaddr` where `bdaddr` is a *bytes* object containing the Bluetooth address in a string format. (ex. `b'12:23:34:45:56:67'`) This protocol is not supported under FreeBSD.
- *AF_ALG* is a Linux-only socket based interface to Kernel cryptography. An algorithm socket is configured with a tuple of two to four elements `(type, name [, feat [, mask]])`, where :
  - *type* is the algorithm type as string, e.g. `aead`, `hash`, `skcipher` or `rng`.
  - *name* is the algorithm name and operation mode as string, e.g. `sha256`, `hmac(sha256)`, `cbc(aes)` or `drbg_nopr_ctr_aes256`.
  - *feat* and *mask* are unsigned 32bit integers.

  Availability : Linux >= 2.6.38.

  Some algorithm types require more recent Kernels.

  Ajouté dans la version 3.6.
- *AF_VSOCK* allows communication between virtual machines and their hosts. The sockets are represented as a `(CID, port)` tuple where the context ID or CID and port are integers.

  Availability : Linux >= 3.9

  See *vsock(7)*

  Ajouté dans la version 3.7.
- *AF_PACKET* is a low-level interface directly to network devices. The addresses are represented by the tuple `(ifname, proto[, pkttype[, hatype[, addr]]])` where :
  - *ifname* - String specifying the device name.
  - *proto* - The Ethernet protocol number. May be *ETH_P_ALL* to capture all protocols, one of the *ETHERTYPE_* constants* or any other Ethernet protocol number.
  - *pkttype* - Optional integer specifying the packet type :
    - `PACKET_HOST` (the default) - Packet addressed to the local host.
    - `PACKET_BROADCAST` - Physical-layer broadcast packet.
    - `PACKET_MULTICAST` - Packet sent to a physical-layer multicast address.
    - `PACKET_OTHERHOST` - Packet to some other host that has been caught by a device driver in promiscuous mode.
    - `PACKET_OUTGOING` - Packet originating from the local host that is looped back to a packet socket.
  - *hatype* - Optional integer specifying the ARP hardware address type.
  - *addr* - Optional bytes-like object specifying the hardware physical address, whose interpretation depends on the device.

  Availability : Linux >= 2.2.
- *AF_QIPCRTR* is a Linux-only socket based interface for communicating with services running on co-processors in Qualcomm platforms. The address family is represented as a `(node, port)` tuple where the *node* and *port* are non-negative integers.

Availability : Linux >= 4.7.

Ajouté dans la version 3.8.

— `IPPROTO_UDPLITE` is a variant of UDP which allows you to specify what portion of a packet is covered with the checksum. It adds two socket options that you can change. `self.setsockopt(IPPROTO_UDPLITE, UDPLITE_SEND_CSCOV, length)` will change what portion of outgoing packets are covered by the checksum and `self.setsockopt(IPPROTO_UDPLITE, UDPLITE_RECV_CSCOV, length)` will filter out packets which cover too little of their data. In both cases `length` should be in `range(8, 2**16, 8)`.

Such a socket should be constructed with `socket(AF_INET, SOCK_DGRAM, IPPROTO_UDPLITE)` for IPv4 or `socket(AF_INET6, SOCK_DGRAM, IPPROTO_UDPLITE)` for IPv6.

Availability : Linux >= 2.6.20, FreeBSD >= 10.1

Ajouté dans la version 3.9.

— *`AF_HYPERV`* is a Windows-only socket based interface for communicating with Hyper-V hosts and guests. The address family is represented as a `(vm_id, service_id)` tuple where the `vm_id` and `service_id` are UUID strings.

The `vm_id` is the virtual machine identifier or a set of known VMID values if the target is not a specific virtual machine. Known VMID constants defined on `socket` are :

— `HV_GUID_ZERO`
— `HV_GUID_BROADCAST`
— `HV_GUID_WILDCARD` - Used to bind on itself and accept connections from all partitions.
— `HV_GUID_CHILDREN` - Used to bind on itself and accept connection from child partitions.
— `HV_GUID_LOOPBACK` - Used as a target to itself.
— `HV_GUID_PARENT` - When used as a bind accepts connection from the parent partition. When used as an address target it will connect to the parent partition.

The `service_id` is the service identifier of the registered service.

Ajouté dans la version 3.12.

If you use a hostname in the *host* portion of IPv4/v6 socket address, the program may show a nondeterministic behavior, as Python uses the first address returned from the DNS resolution. The socket address will be resolved differently into an actual IPv4/v6 address, depending on the results from DNS resolution and/or the host configuration. For deterministic behavior use a numeric address in *host* portion.

All errors raise exceptions. The normal exceptions for invalid argument types and out-of-memory conditions can be raised. Errors related to socket or address semantics raise *`OSError`* or one of its subclasses.

Non-blocking mode is supported through *`setblocking()`*. A generalization of this based on timeouts is supported through *`settimeout()`*.

19.2.2 Module contents

The module *`socket`* exports the following elements.

Exceptions

exception `socket.`**`error`**

A deprecated alias of *`OSError`*.

Modifié dans la version 3.3 : Following **PEP 3151**, this class was made an alias of *`OSError`*.

exception `socket.`**`herror`**

A subclass of *`OSError`*, this exception is raised for address-related errors, i.e. for functions that use *h_errno* in the POSIX C API, including *`gethostbyname_ex()`* and *`gethostbyaddr()`*. The accompanying value is a pair `(h_errno, string)` representing an error returned by a library call. *h_errno* is a numeric value, while *string* represents the description of *h_errno*, as returned by the `hstrerror()` C function.

Modifié dans la version 3.3 : This class was made a subclass of *`OSError`*.

exception `socket.`**`gaierror`**

A subclass of *`OSError`*, this exception is raised for address-related errors by *`getaddrinfo()`* and

getnameinfo(). The accompanying value is a pair `(error, string)` representing an error returned by a library call. *string* represents the description of *error*, as returned by the `gai_strerror()` C function. The numeric *error* value will match one of the `EAI_*` constants defined in this module.

Modifié dans la version 3.3 : This class was made a subclass of *OSError*.

exception socket.**timeout**

A deprecated alias of *TimeoutError*.

A subclass of *OSError*, this exception is raised when a timeout occurs on a socket which has had timeouts enabled via a prior call to *settimeout()* (or implicitly through *setdefaulttimeout()*). The accompanying value is a string whose value is currently always "timed out".

Modifié dans la version 3.3 : This class was made a subclass of *OSError*.

Modifié dans la version 3.10 : cette classe est devenue un alias de *TimeoutError*.

Constantes

The AF_* and SOCK_* constants are now `AddressFamily` and `SocketKind` *IntEnum* collections.

Ajouté dans la version 3.4.

socket.**AF_UNIX**

socket.**AF_INET**

socket.**AF_INET6**

These constants represent the address (and protocol) families, used for the first argument to *socket()*. If the *AF_UNIX* constant is not defined then this protocol is unsupported. More constants may be available depending on the system.

socket.**AF_UNSPEC**

AF_UNSPEC means that *getaddrinfo()* should return socket addresses for any address family (either IPv4, IPv6, or any other) that can be used.

socket.**SOCK_STREAM**

socket.**SOCK_DGRAM**

socket.**SOCK_RAW**

socket.**SOCK_RDM**

socket.**SOCK_SEQPACKET**

These constants represent the socket types, used for the second argument to *socket()*. More constants may be available depending on the system. (Only *SOCK_STREAM* and *SOCK_DGRAM* appear to be generally useful.)

socket.**SOCK_CLOEXEC**

socket.**SOCK_NONBLOCK**

These two constants, if defined, can be combined with the socket types and allow you to set some flags atomically (thus avoiding possible race conditions and the need for separate calls).

> **Voir aussi**
>
> Secure File Descriptor Handling for a more thorough explanation.

Availability : Linux >= 2.6.27.

Ajouté dans la version 3.2.

SO_*

socket.**SOMAXCONN**

MSG_*

SOL_*

SCM_*

IPPROTO_*

IPPORT_*

INADDR_*

IP_*

IPV6_*

EAI_*

AI_*

NI_*

TCP_*

Many constants of these forms, documented in the Unix documentation on sockets and/or the IP protocol, are also defined in the socket module. They are generally used in arguments to the *setsockopt()* and *getsockopt()* methods of socket objects. In most cases, only those symbols that are defined in the Unix header files are defined ; for a few symbols, default values are provided.

Modifié dans la version 3.6 : SO_DOMAIN, SO_PROTOCOL, SO_PEERSEC, SO_PASSSEC, TCP_USER_TIMEOUT, TCP_CONGESTION were added.

Modifié dans la version 3.6.5 : Added support for TCP_FASTOPEN, TCP_KEEPCNT on Windows platforms when available.

Modifié dans la version 3.7 : TCP_NOTSENT_LOWAT was added.

Added support for TCP_KEEPIDLE, TCP_KEEPINTVL on Windows platforms when available.

Modifié dans la version 3.10 : IP_RECVTOS was added. Added TCP_KEEPALIVE. On MacOS this constant can be used in the same way that TCP_KEEPIDLE is used on Linux.

Modifié dans la version 3.11 : Added TCP_CONNECTION_INFO. On MacOS this constant can be used in the same way that TCP_INFO is used on Linux and BSD.

Modifié dans la version 3.12 : Added SO_RTABLE and SO_USER_COOKIE. On OpenBSD and FreeBSD respectively those constants can be used in the same way that SO_MARK is used on Linux. Also added missing TCP socket options from Linux : TCP_MD5SIG, TCP_THIN_LINEAR_TIMEOUTS, TCP_THIN_DUPACK, TCP_REPAIR, TCP_REPAIR_QUEUE, TCP_QUEUE_SEQ, TCP_REPAIR_OPTIONS, TCP_TIMESTAMP, TCP_CC_INFO, TCP_SAVE_SYN, TCP_SAVED_SYN, TCP_REPAIR_WINDOW, TCP_FASTOPEN_CONNECT, TCP_ULP, TCP_MD5SIG_EXT, TCP_FASTOPEN_KEY, TCP_FASTOPEN_NO_COOKIE, TCP_ZEROCOPY_RECEIVE, TCP_INQ, TCP_TX_DELAY. Added IP_PKTINFO, IP_UNBLOCK_SOURCE, IP_BLOCK_SOURCE, IP_ADD_SOURCE_MEMBERSHIP, IP_DROP_SOURCE_MEMBERSHIP.

Modifié dans la version 3.13 : Added SO_BINDTOIFINDEX. On Linux this constant can be used in the same way that SO_BINDTODEVICE is used, but with the index of a network interface instead of its name.

Modifié dans la version 3.14 : Added missing IP_RECVERR, IPV6_RECVERR, IP_RECVTTL, and IP_RECVORIGDSTADDR on Linux.

Modifié dans la version 3.14 : Added support for TCP_QUICKACK on Windows platforms when available.

socket.**AF_CAN**

socket.**PF_CAN**

SOL_CAN_*

CAN_*

Many constants of these forms, documented in the Linux documentation, are also defined in the socket module.

Availability : Linux >= 2.6.25, NetBSD >= 8.

Ajouté dans la version 3.3.

Modifié dans la version 3.11 : NetBSD support was added.

socket.**CAN_BCM**

CAN_BCM_*

CAN_BCM, in the CAN protocol family, is the broadcast manager (BCM) protocol. Broadcast manager constants, documented in the Linux documentation, are also defined in the socket module.

Availability : Linux >= 2.6.25.

> **Note**
>
> The `CAN_BCM_CAN_FD_FRAME` flag is only available on Linux >= 4.8.

Ajouté dans la version 3.4.

`socket.`**CAN_RAW_FD_FRAMES**

Enables CAN FD support in a CAN_RAW socket. This is disabled by default. This allows your application to send both CAN and CAN FD frames ; however, you must accept both CAN and CAN FD frames when reading from the socket.

This constant is documented in the Linux documentation.

Availability : Linux >= 3.6.

Ajouté dans la version 3.5.

`socket.`**CAN_RAW_JOIN_FILTERS**

Joins the applied CAN filters such that only CAN frames that match all given CAN filters are passed to user space.

This constant is documented in the Linux documentation.

Availability : Linux >= 4.1.

Ajouté dans la version 3.9.

`socket.`**CAN_ISOTP**

CAN_ISOTP, in the CAN protocol family, is the ISO-TP (ISO 15765-2) protocol. ISO-TP constants, documented in the Linux documentation.

Availability : Linux >= 2.6.25.

Ajouté dans la version 3.7.

`socket.`**CAN_J1939**

CAN_J1939, in the CAN protocol family, is the SAE J1939 protocol. J1939 constants, documented in the Linux documentation.

Availability : Linux >= 5.4.

Ajouté dans la version 3.9.

`socket.`**AF_DIVERT**

`socket.`**PF_DIVERT**

These two constants, documented in the FreeBSD divert(4) manual page, are also defined in the socket module.

Availability : FreeBSD >= 14.0.

Ajouté dans la version 3.12.

`socket.`**AF_PACKET**

`socket.`**PF_PACKET**

PACKET_*

Many constants of these forms, documented in the Linux documentation, are also defined in the socket module.

Availability : Linux >= 2.2.

socket.**ETH_P_ALL**

ETH_P_ALL can be used in the *socket* constructor as *proto* for the *AF_PACKET* family in order to capture every packet, regardless of protocol.

For more information, see the *packet(7)* manpage.

Availability : Linux.

Ajouté dans la version 3.12.

socket.**AF_RDS**

socket.**PF_RDS**

socket.**SOL_RDS**

RDS_*

Many constants of these forms, documented in the Linux documentation, are also defined in the socket module.

Availability : Linux >= 2.6.30.

Ajouté dans la version 3.3.

socket.**SIO_RCVALL**

socket.**SIO_KEEPALIVE_VALS**

socket.**SIO_LOOPBACK_FAST_PATH**

RCVALL_*

Constants for Windows' WSAIoctl(). The constants are used as arguments to the *ioctl()* method of socket objects.

Modifié dans la version 3.6 : SIO_LOOPBACK_FAST_PATH was added.

TIPC_*

TIPC related constants, matching the ones exported by the C socket API. See the TIPC documentation for more information.

socket.**AF_ALG**

socket.**SOL_ALG**

ALG_*

Constants for Linux Kernel cryptography.

Availability : Linux >= 2.6.38.

Ajouté dans la version 3.6.

socket.**AF_VSOCK**

socket.**IOCTL_VM_SOCKETS_GET_LOCAL_CID**

VMADDR*

SO_VM*

Constants for Linux host/guest communication.

Availability : Linux >= 4.8.

Ajouté dans la version 3.7.

socket.**AF_LINK**

Availability : BSD, macOS.

Ajouté dans la version 3.4.

socket.**has_ipv6**

This constant contains a boolean value which indicates if IPv6 is supported on this platform.

socket.**BDADDR_ANY**

socket.**BDADDR_LOCAL**

These are string constants containing Bluetooth addresses with special meanings. For example, *BDADDR_ANY* can be used to indicate any address when specifying the binding socket with BTPROTO_RFCOMM.

socket.**HCI_FILTER**

socket.**HCI_TIME_STAMP**

socket.**HCI_DATA_DIR**

For use with BTPROTO_HCI. *HCI_FILTER* is not available for NetBSD or DragonFlyBSD. *HCI_TIME_STAMP* and *HCI_DATA_DIR* are not available for FreeBSD, NetBSD, or DragonFlyBSD.

socket.**AF_QIPCRTR**

Constant for Qualcomm's IPC router protocol, used to communicate with service providing remote processors.

Availability : Linux >= 4.7.

socket.**SCM_CREDS2**

socket.**LOCAL_CREDS**

socket.**LOCAL_CREDS_PERSISTENT**

LOCAL_CREDS and LOCAL_CREDS_PERSISTENT can be used with SOCK_DGRAM, SOCK_STREAM sockets, equivalent to Linux/DragonFlyBSD SO_PASSCRED, while LOCAL_CREDS sends the credentials at first read, LOCAL_CREDS_PERSISTENT sends for each read, SCM_CREDS2 must be then used for the latter for the message type.

Ajouté dans la version 3.11.

Availability : FreeBSD.

socket.**SO_INCOMING_CPU**

Constant to optimize CPU locality, to be used in conjunction with SO_REUSEPORT.

Ajouté dans la version 3.11.

Availability : Linux >= 3.9

socket.**SO_REUSEPORT_LB**

Constant to enable duplicate address and port bindings with load balancing.

Ajouté dans la version 3.14.

Availability : FreeBSD >= 12.0

socket.**AF_HYPERV**

socket.**HV_PROTOCOL_RAW**

socket.**HVSOCKET_CONNECT_TIMEOUT**

socket.**HVSOCKET_CONNECT_TIMEOUT_MAX**

socket.**HVSOCKET_CONNECTED_SUSPEND**

socket.**HVSOCKET_ADDRESS_FLAG_PASSTHRU**

socket.**HV_GUID_ZERO**

socket.**HV_GUID_WILDCARD**

socket.**HV_GUID_BROADCAST**

socket.**HV_GUID_CHILDREN**

socket.**HV_GUID_LOOPBACK**

socket.**HV_GUID_PARENT**

Constants for Windows Hyper-V sockets for host/guest communications.

Availability : Windows.

Ajouté dans la version 3.12.

socket.**ETHERTYPE_ARP**

socket.**ETHERTYPE_IP**

socket.**ETHERTYPE_IPV6**

socket.**ETHERTYPE_VLAN**

IEEE 802.3 protocol number. constants.

Availability : Linux, FreeBSD, macOS.

Ajouté dans la version 3.12.

socket.**SHUT_RD**

socket.**SHUT_WR**

socket.**SHUT_RDWR**

These constants are used by the `shutdown()` method of socket objects.

Availability : not WASI.

Fonctions

Creating sockets

The following functions all create *socket objects*.

class socket.**socket**(*family=AF_INET*, *type=SOCK_STREAM*, *proto=0*, *fileno=None*)

Create a new socket using the given address family, socket type and protocol number. The address family should be `AF_INET` (the default), `AF_INET6`, `AF_UNIX`, `AF_CAN`, `AF_PACKET`, or `AF_RDS`. The socket type should be `SOCK_STREAM` (the default), `SOCK_DGRAM`, `SOCK_RAW` or perhaps one of the other `SOCK_` constants. The protocol number is usually zero and may be omitted or in the case where the address family is `AF_CAN` the protocol should be one of `CAN_RAW`, `CAN_BCM`, `CAN_ISOTP` or `CAN_J1939`.

If *fileno* is specified, the values for *family*, *type*, and *proto* are auto-detected from the specified file descriptor. Auto-detection can be overruled by calling the function with explicit *family*, *type*, or *proto* arguments. This only affects how Python represents e.g. the return value of `socket.getpeername()` but not the actual OS resource. Unlike `socket.fromfd()`, *fileno* will return the same socket and not a duplicate. This may help close a detached socket using `socket.close()`.

Il n'est *pas possible d'hériter* du connecteur nouvellement créé.

Raises an *auditing event* `socket.__new__` with arguments `self`, `family`, `type`, `protocol`.

Modifié dans la version 3.3 : The AF_CAN family was added. The AF_RDS family was added.

Modifié dans la version 3.4 : The CAN_BCM protocol was added.

Modifié dans la version 3.4 : The returned socket is now non-inheritable.

Modifié dans la version 3.7 : The CAN_ISOTP protocol was added.

Modifié dans la version 3.7 : When `SOCK_NONBLOCK` or `SOCK_CLOEXEC` bit flags are applied to *type* they are cleared, and `socket.type` will not reflect them. They are still passed to the underlying system `socket()` call. Therefore,

```
sock = socket.socket(
    socket.AF_INET,
    socket.SOCK_STREAM | socket.SOCK_NONBLOCK)
```

will still create a non-blocking socket on OSes that support `SOCK_NONBLOCK`, but `sock.type` will be set to `socket.SOCK_STREAM`.

Modifié dans la version 3.9 : The CAN_J1939 protocol was added.

Modifié dans la version 3.10 : The IPPROTO_MPTCP protocol was added.

socket.**socketpair**([*family*[, *type*[, *proto*]]])

Build a pair of connected socket objects using the given address family, socket type, and protocol number. Address family, socket type, and protocol number are as for the `socket()` function above. The default family is `AF_UNIX` if defined on the platform ; otherwise, the default is `AF_INET`.

The newly created sockets are *non-inheritable*.

Modifié dans la version 3.2 : The returned socket objects now support the whole socket API, rather than a subset.

Modifié dans la version 3.4 : The returned sockets are now non-inheritable.

Modifié dans la version 3.5 : Windows support added.

`socket.create_connection`(*address*, *timeout=GLOBAL_DEFAULT*, *source_address=None*, *, *all_errors=False*)

Connect to a TCP service listening on the internet *address* (a 2-tuple `(host, port)`), and return the socket object. This is a higher-level function than `socket.connect()` : if *host* is a non-numeric hostname, it will try to resolve it for both `AF_INET` and `AF_INET6`, and then try to connect to all possible addresses in turn until a connection succeeds. This makes it easy to write clients that are compatible to both IPv4 and IPv6.

Passing the optional *timeout* parameter will set the timeout on the socket instance before attempting to connect. If no *timeout* is supplied, the global default timeout setting returned by `getdefaulttimeout()` is used.

If supplied, *source_address* must be a 2-tuple `(host, port)` for the socket to bind to as its source address before connecting. If host or port are '' or 0 respectively the OS default behavior will be used.

When a connection cannot be created, an exception is raised. By default, it is the exception from the last address in the list. If *all_errors* is `True`, it is an `ExceptionGroup` containing the errors of all attempts.

Modifié dans la version 3.2 : Le paramètre *source_address* a été ajouté.

Modifié dans la version 3.11 : *all_errors* was added.

`socket.create_server`(*address*, *, *family=AF_INET*, *backlog=None*, *reuse_port=False*, *dualstack_ipv6=False*)

Convenience function which creates a TCP socket bound to *address* (a 2-tuple `(host, port)`) and returns the socket object.

family should be either `AF_INET` or `AF_INET6`. *backlog* is the queue size passed to `socket.listen()` ; if not specified , a default reasonable value is chosen. *reuse_port* dictates whether to set the `SO_REUSEPORT` socket option.

If *dualstack_ipv6* is true and the platform supports it the socket will be able to accept both IPv4 and IPv6 connections, else it will raise `ValueError`. Most POSIX platforms and Windows are supposed to support this functionality. When this functionality is enabled the address returned by `socket.getpeername()` when an IPv4 connection occurs will be an IPv6 address represented as an IPv4-mapped IPv6 address. If *dualstack_ipv6* is false it will explicitly disable this functionality on platforms that enable it by default (e.g. Linux). This parameter can be used in conjunction with `has_dualstack_ipv6()` :

```
import socket

addr = ("", 8080)  # all interfaces, port 8080
if socket.has_dualstack_ipv6():
    s = socket.create_server(addr, family=socket.AF_INET6, dualstack_ipv6=True)
else:
    s = socket.create_server(addr)
```

Note

On POSIX platforms the `SO_REUSEADDR` socket option is set in order to immediately reuse previous sockets which were bound on the same *address* and remained in TIME_WAIT state.

Ajouté dans la version 3.8.

`socket.has_dualstack_ipv6`()

Return `True` if the platform supports creating a TCP socket which can handle both IPv4 and IPv6 connections.

Ajouté dans la version 3.8.

socket.**fromfd**(*fd*, *family*, *type*, *proto=0*)

Duplicate the file descriptor *fd* (an integer as returned by a file object's `fileno()` method) and build a socket object from the result. Address family, socket type and protocol number are as for the `socket()` function above. The file descriptor should refer to a socket, but this is not checked --- subsequent operations on the object may fail if the file descriptor is invalid. This function is rarely needed, but can be used to get or set socket options on a socket passed to a program as standard input or output (such as a server started by the Unix inet daemon). The socket is assumed to be in blocking mode.

Il n'est *pas possible d'hériter* du connecteur nouvellement créé.

Modifié dans la version 3.4 : The returned socket is now non-inheritable.

socket.**fromshare**(*data*)

Instantiate a socket from data obtained from the `socket.share()` method. The socket is assumed to be in blocking mode.

Availability : Windows.

Ajouté dans la version 3.3.

socket.**SocketType**

This is a Python type object that represents the socket object type. It is the same as `type(socket(...))`.

Autres fonctions

The `socket` module also offers various network-related services :

socket.**close**(*fd*)

Close a socket file descriptor. This is like `os.close()`, but for sockets. On some platforms (most noticeable Windows) `os.close()` does not work for socket file descriptors.

Ajouté dans la version 3.7.

socket.**getaddrinfo**(*host*, *port*, *family=AF_UNSPEC*, *type=0*, *proto=0*, *flags=0*)

This function wraps the C function `getaddrinfo` of the underlying system.

Translate the *host*/*port* argument into a sequence of 5-tuples that contain all the necessary arguments for creating a socket connected to that service. *host* is a domain name, a string representation of an IPv4/v6 address or `None`. *port* is a string service name such as `'http'`, a numeric port number or `None`. By passing `None` as the value of *host* and *port*, you can pass `NULL` to the underlying C API.

The *family*, *type* and *proto* arguments can be optionally specified in order to provide options and limit the list of addresses returned. Pass their default values (`AF_UNSPEC`, 0, and 0, respectively) to not limit the results. See the note below for details.

The *flags* argument can be one or several of the `AI_*` constants, and will influence how results are computed and returned. For example, `AI_NUMERICHOST` will disable domain name resolution and will raise an error if *host* is a domain name.

The function returns a list of 5-tuples with the following structure :

```
(family, type, proto, canonname, sockaddr)
```

In these tuples, *family*, *type*, *proto* are all integers and are meant to be passed to the `socket()` function. *canonname* will be a string representing the canonical name of the *host* if `AI_CANONNAME` is part of the *flags* argument ; else *canonname* will be empty. *sockaddr* is a tuple describing a socket address, whose format depends on the returned *family* (a `(address, port)` 2-tuple for `AF_INET`, a `(address, port, flowinfo, scope_id)` 4-tuple for `AF_INET6`), and is meant to be passed to the `socket.connect()` method.

> **Note**
>
> If you intend to use results from `getaddrinfo()` to create a socket (rather than, for example, retrieve *canonname*), consider limiting the results by *type* (e.g. `SOCK_STREAM` or `SOCK_DGRAM`) and/or *proto* (e.g.

IPPROTO_TCP or IPPROTO_UDP) that your application can handle.

The behavior with default values of *family*, *type*, *proto* and *flags* is system-specific.

Many systems (for example, most Linux configurations) will return a sorted list of all matching addresses. These addresses should generally be tried in order until a connection succeeds (possibly tried in parallel, for example, using a Happy Eyeballs algorithm). In these cases, limiting the *type* and/or *proto* can help eliminate unsuccessful or unusable connection attempts.

Some systems will, however, only return a single address. (For example, this was reported on Solaris and AIX configurations.) On these systems, limiting the *type* and/or *proto* helps ensure that this address is usable.

Raises an *auditing event* `socket.getaddrinfo` with arguments `host`, `port`, `family`, `type`, `protocol`.

The following example fetches address information for a hypothetical TCP connection to `example.org` on port 80 (results may differ on your system if IPv6 isn't enabled) :

```
>>> socket.getaddrinfo("example.org", 80, proto=socket.IPPROTO_TCP)
[(socket.AF_INET6, socket.SOCK_STREAM,
 6, '', ('2606:2800:220:1:248:1893:25c8:1946', 80, 0, 0)),
 (socket.AF_INET, socket.SOCK_STREAM,
 6, '', ('93.184.216.34', 80))]
```

Modifié dans la version 3.2 : parameters can now be passed using keyword arguments.

Modifié dans la version 3.7 : for IPv6 multicast addresses, string representing an address will not contain `%scope_id` part.

`socket.`**`getfqdn`**`(`[*name*]`)`

Return a fully qualified domain name for *name*. If *name* is omitted or empty, it is interpreted as the local host. To find the fully qualified name, the hostname returned by `gethostbyaddr()` is checked, followed by aliases for the host, if available. The first name which includes a period is selected. In case no fully qualified domain name is available and *name* was provided, it is returned unchanged. If *name* was empty or equal to `'0.0.0.0'`, the hostname from `gethostname()` is returned.

`socket.`**`gethostbyname`**`(`*hostname*`)`

Translate a host name to IPv4 address format. The IPv4 address is returned as a string, such as `'100.50.200.5'`. If the host name is an IPv4 address itself it is returned unchanged. See `gethostbyname_ex()` for a more complete interface. `gethostbyname()` does not support IPv6 name resolution, and `getaddrinfo()` should be used instead for IPv4/v6 dual stack support.

Raises an *auditing event* `socket.gethostbyname` with argument `hostname`.

Availability : not WASI.

`socket.`**`gethostbyname_ex`**`(`*hostname*`)`

Translate a host name to IPv4 address format, extended interface. Return a 3-tuple `(hostname, aliaslist, ipaddrlist)` where *hostname* is the host's primary host name, *aliaslist* is a (possibly empty) list of alternative host names for the same address, and *ipaddrlist* is a list of IPv4 addresses for the same interface on the same host (often but not always a single address). `gethostbyname_ex()` does not support IPv6 name resolution, and `getaddrinfo()` should be used instead for IPv4/v6 dual stack support.

Raises an *auditing event* `socket.gethostbyname` with argument `hostname`.

Availability : not WASI.

`socket.`**`gethostname`**`()`

Return a string containing the hostname of the machine where the Python interpreter is currently executing.

Raises an *auditing event* `socket.gethostname` with no arguments.

Note : `gethostname()` doesn't always return the fully qualified domain name ; use `getfqdn()` for that.

Availability : not WASI.

socket.**gethostbyaddr**(*ip_address*)

Return a 3-tuple `(hostname, aliaslist, ipaddrlist)` where *hostname* is the primary host name responding to the given *ip_address*, *aliaslist* is a (possibly empty) list of alternative host names for the same address, and *ipaddrlist* is a list of IPv4/v6 addresses for the same interface on the same host (most likely containing only a single address). To find the fully qualified domain name, use the function `getfqdn()`. `gethostbyaddr()` supports both IPv4 and IPv6.

Raises an *auditing event* `socket.gethostbyaddr` with argument `ip_address`.

Availability : not WASI.

socket.**getnameinfo**(*sockaddr*, *flags*)

Translate a socket address *sockaddr* into a 2-tuple `(host, port)`. Depending on the settings of *flags*, the result can contain a fully qualified domain name or numeric address representation in *host*. Similarly, *port* can contain a string port name or a numeric port number.

For IPv6 addresses, `%scope_id` is appended to the host part if *sockaddr* contains meaningful *scope_id*. Usually this happens for multicast addresses.

For more information about *flags* you can consult `getnameinfo(3)`.

Raises an *auditing event* `socket.getnameinfo` with argument `sockaddr`.

Availability : not WASI.

socket.**getprotobyname**(*protocolname*)

Translate an internet protocol name (for example, `'icmp'`) to a constant suitable for passing as the (optional) third argument to the `socket()` function. This is usually only needed for sockets opened in "raw" mode (`SOCK_RAW`); for the normal socket modes, the correct protocol is chosen automatically if the protocol is omitted or zero.

Availability : not WASI.

socket.**getservbyname**(*servicename*[, *protocolname*])

Translate an internet service name and protocol name to a port number for that service. The optional protocol name, if given, should be `'tcp'` or `'udp'`, otherwise any protocol will match.

Raises an *auditing event* `socket.getservbyname` with arguments `servicename`, `protocolname`.

Availability : not WASI.

socket.**getservbyport**(*port*[, *protocolname*])

Translate an internet port number and protocol name to a service name for that service. The optional protocol name, if given, should be `'tcp'` or `'udp'`, otherwise any protocol will match.

Raises an *auditing event* `socket.getservbyport` with arguments `port`, `protocolname`.

Availability : not WASI.

socket.**ntohl**(*x*)

Convert 32-bit positive integers from network to host byte order. On machines where the host byte order is the same as network byte order, this is a no-op; otherwise, it performs a 4-byte swap operation.

socket.**ntohs**(*x*)

Convert 16-bit positive integers from network to host byte order. On machines where the host byte order is the same as network byte order, this is a no-op; otherwise, it performs a 2-byte swap operation.

Modifié dans la version 3.10 : Raises `OverflowError` if *x* does not fit in a 16-bit unsigned integer.

socket.**htonl**(*x*)

Convert 32-bit positive integers from host to network byte order. On machines where the host byte order is the same as network byte order, this is a no-op; otherwise, it performs a 4-byte swap operation.

socket.**htons**(*x*)

Convert 16-bit positive integers from host to network byte order. On machines where the host byte order is the same as network byte order, this is a no-op; otherwise, it performs a 2-byte swap operation.

Modifié dans la version 3.10 : Raises `OverflowError` if *x* does not fit in a 16-bit unsigned integer.

`socket.inet_aton`(*ip_string*)

Convert an IPv4 address from dotted-quad string format (for example, '123.45.67.89') to 32-bit packed binary format, as a bytes object four characters in length. This is useful when conversing with a program that uses the standard C library and needs objects of type `in_addr`, which is the C type for the 32-bit packed binary this function returns.

`inet_aton()` also accepts strings with less than three dots ; see the Unix manual page `inet(3)` for details.

If the IPv4 address string passed to this function is invalid, `OSError` will be raised. Note that exactly what is valid depends on the underlying C implementation of `inet_aton()`.

`inet_aton()` does not support IPv6, and `inet_pton()` should be used instead for IPv4/v6 dual stack support.

`socket.inet_ntoa`(*packed_ip*)

Convert a 32-bit packed IPv4 address (a *bytes-like object* four bytes in length) to its standard dotted-quad string representation (for example, '123.45.67.89'). This is useful when conversing with a program that uses the standard C library and needs objects of type `in_addr`, which is the C type for the 32-bit packed binary data this function takes as an argument.

If the byte sequence passed to this function is not exactly 4 bytes in length, `OSError` will be raised. `inet_ntoa()` does not support IPv6, and `inet_ntop()` should be used instead for IPv4/v6 dual stack support.

Modifié dans la version 3.5 : N'importe quel *bytes-like object* est maintenant accepté.

`socket.inet_pton`(*address_family*, *ip_string*)

Convert an IP address from its family-specific string format to a packed, binary format. `inet_pton()` is useful when a library or network protocol calls for an object of type `in_addr` (similar to `inet_aton()`) or `in6_addr`.

Supported values for *address_family* are currently `AF_INET` and `AF_INET6`. If the IP address string *ip_string* is invalid, `OSError` will be raised. Note that exactly what is valid depends on both the value of *address_family* and the underlying implementation of `inet_pton()`.

Availability : Unix, Windows.

Modifié dans la version 3.4 : Ajout de la gestion de Windows.

`socket.inet_ntop`(*address_family*, *packed_ip*)

Convert a packed IP address (a *bytes-like object* of some number of bytes) to its standard, family-specific string representation (for example, `'7.10.0.5'` or `'5aef:2b::8'`). `inet_ntop()` is useful when a library or network protocol returns an object of type `in_addr` (similar to `inet_ntoa()`) or `in6_addr`.

Supported values for *address_family* are currently `AF_INET` and `AF_INET6`. If the bytes object *packed_ip* is not the correct length for the specified address family, `ValueError` will be raised. `OSError` is raised for errors from the call to `inet_ntop()`.

Availability : Unix, Windows.

Modifié dans la version 3.4 : Ajout de la gestion de Windows.

Modifié dans la version 3.5 : N'importe quel *bytes-like object* est maintenant accepté.

`socket.CMSG_LEN`(*length*)

Return the total length, without trailing padding, of an ancillary data item with associated data of the given *length*. This value can often be used as the buffer size for `recvmsg()` to receive a single item of ancillary data, but **RFC 3542** requires portable applications to use `CMSG_SPACE()` and thus include space for padding, even when the item will be the last in the buffer. Raises `OverflowError` if *length* is outside the permissible range of values.

Availability : Unix, not WASI.

Most Unix platforms.

Ajouté dans la version 3.3.

`socket.CMSG_SPACE(length)`

Return the buffer size needed for `recvmsg()` to receive an ancillary data item with associated data of the given *length*, along with any trailing padding. The buffer space needed to receive multiple items is the sum of the `CMSG_SPACE()` values for their associated data lengths. Raises `OverflowError` if *length* is outside the permissible range of values.

Note that some systems might support ancillary data without providing this function. Also note that setting the buffer size using the results of this function may not precisely limit the amount of ancillary data that can be received, since additional data may be able to fit into the padding area.

Availability : Unix, not WASI.

most Unix platforms.

Ajouté dans la version 3.3.

`socket.getdefaulttimeout()`

Return the default timeout in seconds (float) for new socket objects. A value of `None` indicates that new socket objects have no timeout. When the socket module is first imported, the default is `None`.

`socket.setdefaulttimeout(timeout)`

Set the default timeout in seconds (float) for new socket objects. When the socket module is first imported, the default is `None`. See `settimeout()` for possible values and their respective meanings.

`socket.sethostname(name)`

Set the machine's hostname to *name*. This will raise an `OSError` if you don't have enough rights.

Raises an *auditing event* `socket.sethostname` with argument `name`.

Availability : Unix, not Android.

Ajouté dans la version 3.3.

`socket.if_nameindex()`

Return a list of network interface information (index int, name string) tuples. `OSError` if the system call fails.

Availability : Unix, Windows, not WASI.

Ajouté dans la version 3.3.

Modifié dans la version 3.8 : Windows support was added.

> **Note**
>
> On Windows network interfaces have different names in different contexts (all names are examples) :
> — UUID : `{FB605B73-AAC2-49A6-9A2F-25416AEA0573}`
> — name : `ethernet_32770`
> — friendly name : `vEthernet (nat)`
> — description : `Hyper-V Virtual Ethernet Adapter`
>
> This function returns names of the second form from the list, `ethernet_32770` in this example case.

`socket.if_nametoindex(if_name)`

Return a network interface index number corresponding to an interface name. `OSError` if no interface with the given name exists.

Availability : Unix, Windows, not WASI.

Ajouté dans la version 3.3.

Modifié dans la version 3.8 : Windows support was added.

> **Voir aussi**
>
> "Interface name" is a name as documented in `if_nameindex()`.

`socket.`**`if_indextoname`**(*if_index*)

Return a network interface name corresponding to an interface index number. `OSError` if no interface with the given index exists.

Availability : Unix, Windows, not WASI.

Ajouté dans la version 3.3.

Modifié dans la version 3.8 : Windows support was added.

> **Voir aussi**
>
> "Interface name" is a name as documented in `if_nameindex()`.

`socket.`**`send_fds`**(*sock*, *buffers*, *fds*[, *flags*[, *address*]])

Send the list of file descriptors *fds* over an `AF_UNIX` socket *sock*. The *fds* parameter is a sequence of file descriptors. Consult `sendmsg()` for the documentation of these parameters.

Availability : Unix, Windows, not WASI.

Unix platforms supporting `sendmsg()` and `SCM_RIGHTS` mechanism.

Ajouté dans la version 3.9.

`socket.`**`recv_fds`**(*sock*, *bufsize*, *maxfds*[, *flags*])

Receive up to *maxfds* file descriptors from an `AF_UNIX` socket *sock*. Return `(msg, list(fds), flags, addr)`. Consult `recvmsg()` for the documentation of these parameters.

Availability : Unix, Windows, not WASI.

Unix platforms supporting `sendmsg()` and `SCM_RIGHTS` mechanism.

Ajouté dans la version 3.9.

> **Note**
>
> Any truncated integers at the end of the list of file descriptors.

19.2.3 Socket Objects

Socket objects have the following methods. Except for `makefile()`, these correspond to Unix system calls applicable to sockets.

Modifié dans la version 3.2 : Support for the *context manager* protocol was added. Exiting the context manager is equivalent to calling `close()`.

`socket.`**`accept`**()

Accept a connection. The socket must be bound to an address and listening for connections. The return value is a pair `(conn, address)` where *conn* is a *new* socket object usable to send and receive data on the connection, and *address* is the address bound to the socket on the other end of the connection.

Il n'est *pas possible d'hériter* du connecteur nouvellement créé.

Modifié dans la version 3.4 : The socket is now non-inheritable.

Modifié dans la version 3.5 : Si l'appel système est interrompu et que le gestionnaire de signal ne lève aucune exception, la fonction réessaye l'appel système au lieu de lever une `InterruptedError` (voir la **PEP 475** à propos du raisonnement).

socket.**bind**(*address*)

Bind the socket to *address*. The socket must not already be bound. (The format of *address* depends on the address family --- see above.)

Raises an *auditing event* `socket.bind` with arguments `self`, `address`.

Availability : not WASI.

socket.**close**()

Mark the socket closed. The underlying system resource (e.g. a file descriptor) is also closed when all file objects from `makefile()` are closed. Once that happens, all future operations on the socket object will fail. The remote end will receive no more data (after queued data is flushed).

Sockets are automatically closed when they are garbage-collected, but it is recommended to `close()` them explicitly, or to use a `with` statement around them.

Modifié dans la version 3.6 : `OSError` is now raised if an error occurs when the underlying `close()` call is made.

> **Note**
>
> `close()` releases the resource associated with a connection but does not necessarily close the connection immediately. If you want to close the connection in a timely fashion, call `shutdown()` before `close()`.

socket.**connect**(*address*)

Connect to a remote socket at *address*. (The format of *address* depends on the address family --- see above.)

If the connection is interrupted by a signal, the method waits until the connection completes, or raise a `TimeoutError` on timeout, if the signal handler doesn't raise an exception and the socket is blocking or has a timeout. For non-blocking sockets, the method raises an `InterruptedError` exception if the connection is interrupted by a signal (or the exception raised by the signal handler).

Raises an *auditing event* `socket.connect` with arguments `self`, `address`.

Modifié dans la version 3.5 : The method now waits until the connection completes instead of raising an `InterruptedError` exception if the connection is interrupted by a signal, the signal handler doesn't raise an exception and the socket is blocking or has a timeout (see the **PEP 475** for the rationale).

Availability : not WASI.

socket.**connect_ex**(*address*)

Like `connect(address)`, but return an error indicator instead of raising an exception for errors returned by the C-level `connect()` call (other problems, such as "host not found," can still raise exceptions). The error indicator is `0` if the operation succeeded, otherwise the value of the `errno` variable. This is useful to support, for example, asynchronous connects.

Raises an *auditing event* `socket.connect` with arguments `self`, `address`.

Availability : not WASI.

socket.**detach**()

Put the socket object into closed state without actually closing the underlying file descriptor. The file descriptor is returned, and can be reused for other purposes.

Ajouté dans la version 3.2.

socket.**dup**()

Duplicate the socket.

Il n'est *pas possible d'hériter* du connecteur nouvellement créé.

Modifié dans la version 3.4 : The socket is now non-inheritable.

Availability : not WASI.

socket.**fileno**()

Return the socket's file descriptor (a small integer), or -1 on failure. This is useful with `select.select()`.

Under Windows the small integer returned by this method cannot be used where a file descriptor can be used (such as `os.fdopen()`). Unix does not have this limitation.

socket.**get_inheritable**()

Get the *inheritable flag* of the socket's file descriptor or socket's handle : `True` if the socket can be inherited in child processes, `False` if it cannot.

Ajouté dans la version 3.4.

socket.**getpeername**()

Return the remote address to which the socket is connected. This is useful to find out the port number of a remote IPv4/v6 socket, for instance. (The format of the address returned depends on the address family --- see above.) On some systems this function is not supported.

socket.**getsockname**()

Return the socket's own address. This is useful to find out the port number of an IPv4/v6 socket, for instance. (The format of the address returned depends on the address family --- see above.)

socket.**getsockopt**(*level*, *optname*[, *buflen*])

Return the value of the given socket option (see the Unix man page `getsockopt(2)`). The needed symbolic constants (*SO_* etc.*) are defined in this module. If *buflen* is absent, an integer option is assumed and its integer value is returned by the function. If *buflen* is present, it specifies the maximum length of the buffer used to receive the option in, and this buffer is returned as a bytes object. It is up to the caller to decode the contents of the buffer (see the optional built-in module `struct` for a way to decode C structures encoded as byte strings).

Availability : not WASI.

socket.**getblocking**()

Return `True` if socket is in blocking mode, `False` if in non-blocking.

This is equivalent to checking `socket.gettimeout() != 0`.

Ajouté dans la version 3.7.

socket.**gettimeout**()

Return the timeout in seconds (float) associated with socket operations, or `None` if no timeout is set. This reflects the last call to `setblocking()` or `settimeout()`.

socket.**ioctl**(*control*, *option*)

The `ioctl()` method is a limited interface to the WSAIoctl system interface. Please refer to the Win32 documentation for more information.

On other platforms, the generic `fcntl.fcntl()` and `fcntl.ioctl()` functions may be used ; they accept a socket object as their first argument.

Currently only the following control codes are supported : `SIO_RCVALL`, `SIO_KEEPALIVE_VALS`, and `SIO_LOOPBACK_FAST_PATH`.

Availability : Windows

Modifié dans la version 3.6 : `SIO_LOOPBACK_FAST_PATH` was added.

socket.**listen**([*backlog*])

Enable a server to accept connections. If *backlog* is specified, it must be at least 0 (if it is lower, it is set to 0) ; it specifies the number of unaccepted connections that the system will allow before refusing new connections. If not specified, a default reasonable value is chosen.

Availability : not WASI.

Modifié dans la version 3.5 : The *backlog* parameter is now optional.

socket.**makefile**(*mode='r'*, *buffering=None*, **, encoding=None*, *errors=None*, *newline=None*)

Return a *file object* associated with the socket. The exact returned type depends on the arguments given to `makefile()`. These arguments are interpreted the same way as by the built-in `open()` function, except the only supported *mode* values are `'r'` (default), `'w'`, `'b'`, or a combination of those.

The socket must be in blocking mode ; it can have a timeout, but the file object's internal buffer may end up in an inconsistent state if a timeout occurs.

Closing the file object returned by `makefile()` won't close the original socket unless all other file objects have been closed and `socket.close()` has been called on the socket object.

> **Note**
>
> On Windows, the file-like object created by `makefile()` cannot be used where a file object with a file descriptor is expected, such as the stream arguments of `subprocess.Popen()`.

socket.**recv**(*bufsize*[, *flags*])

Receive data from the socket. The return value is a bytes object representing the data received. The maximum amount of data to be received at once is specified by *bufsize*. A returned empty bytes object indicates that the client has disconnected. See the Unix manual page `recv(2)` for the meaning of the optional argument *flags* ; it defaults to zero.

Modifié dans la version 3.5 : Si l'appel système est interrompu et que le gestionnaire de signal ne lève aucune exception, la fonction réessaye l'appel système au lieu de lever une `InterruptedError` (voir la **PEP 475** à propos du raisonnement).

socket.**recvfrom**(*bufsize*[, *flags*])

Receive data from the socket. The return value is a pair `(bytes, address)` where *bytes* is a bytes object representing the data received and *address* is the address of the socket sending the data. See the Unix manual page `recv(2)` for the meaning of the optional argument *flags* ; it defaults to zero. (The format of *address* depends on the address family --- see above.)

Modifié dans la version 3.5 : Si l'appel système est interrompu et que le gestionnaire de signal ne lève aucune exception, la fonction réessaye l'appel système au lieu de lever une `InterruptedError` (voir la **PEP 475** à propos du raisonnement).

Modifié dans la version 3.7 : For multicast IPv6 address, first item of *address* does not contain `%scope_id` part anymore. In order to get full IPv6 address use `getnameinfo()`.

socket.**recvmsg**(*bufsize*[, *ancbufsize*[, *flags*]])

Receive normal data (up to *bufsize* bytes) and ancillary data from the socket. The *ancbufsize* argument sets the size in bytes of the internal buffer used to receive the ancillary data ; it defaults to 0, meaning that no ancillary data will be received. Appropriate buffer sizes for ancillary data can be calculated using `CMSG_SPACE()` or `CMSG_LEN()`, and items which do not fit into the buffer might be truncated or discarded. The *flags* argument defaults to 0 and has the same meaning as for `recv()`.

The return value is a 4-tuple : `(data, ancdata, msg_flags, address)`. The *data* item is a `bytes` object holding the non-ancillary data received. The *ancdata* item is a list of zero or more tuples `(cmsg_level, cmsg_type, cmsg_data)` representing the ancillary data (control messages) received : *cmsg_level* and *cmsg_type* are integers specifying the protocol level and protocol-specific type respectively, and *cmsg_data* is a `bytes` object holding the associated data. The *msg_flags* item is the bitwise OR of various flags indicating conditions on the received message ; see your system documentation for details. If the receiving socket is unconnected, *address* is the address of the sending socket, if available ; otherwise, its value is unspecified.

On some systems, `sendmsg()` and `recvmsg()` can be used to pass file descriptors between processes over an `AF_UNIX` socket. When this facility is used (it is often restricted to `SOCK_STREAM` sockets), `recvmsg()` will return, in its ancillary data, items of the form `(socket.SOL_SOCKET, socket.SCM_RIGHTS, fds)`, where *fds* is a `bytes` object representing the new file descriptors as a binary array of the native C `int` type. If `recvmsg()` raises an exception after the system call returns, it will first attempt to close any file descriptors received via this mechanism.

Some systems do not indicate the truncated length of ancillary data items which have been only partially received. If an item appears to extend beyond the end of the buffer, `recvmsg()` will issue a `RuntimeWarning`, and will return the part of it which is inside the buffer provided it has not been truncated before the start of its associated data.

On systems which support the `SCM_RIGHTS` mechanism, the following function will receive up to *maxfds* file descriptors, returning the message data and a list containing the descriptors (while ignoring unexpected conditions such as unrelated control messages being received). See also `sendmsg()`.

```
import socket, array

def recv_fds(sock, msglen, maxfds):
    fds = array.array("i")   # Array of ints
    msg, ancdata, flags, addr = sock.recvmsg(msglen, socket.CMSG_LEN(maxfds *␣
↪fds.itemsize))
    for cmsg_level, cmsg_type, cmsg_data in ancdata:
        if cmsg_level == socket.SOL_SOCKET and cmsg_type == socket.SCM_RIGHTS:
            # Append data, ignoring any truncated integers at the end.
            fds.frombytes(cmsg_data[:len(cmsg_data) - (len(cmsg_data) % fds.
↪itemsize)])
    return msg, list(fds)
```

Availability : Unix.

Most Unix platforms.

Ajouté dans la version 3.3.

Modifié dans la version 3.5 : Si l'appel système est interrompu et que le gestionnaire de signal ne lève aucune exception, la fonction réessaye l'appel système au lieu de lever une `InterruptedError` (voir la **PEP 475** à propos du raisonnement).

`socket.`**`recvmsg_into`**(*buffers*[, *ancbufsize*[, *flags*]])

Receive normal data and ancillary data from the socket, behaving as `recvmsg()` would, but scatter the non-ancillary data into a series of buffers instead of returning a new bytes object. The *buffers* argument must be an iterable of objects that export writable buffers (e.g. `bytearray` objects) ; these will be filled with successive chunks of the non-ancillary data until it has all been written or there are no more buffers. The operating system may set a limit (`sysconf()` value `SC_IOV_MAX`) on the number of buffers that can be used. The *ancbufsize* and *flags* arguments have the same meaning as for `recvmsg()`.

The return value is a 4-tuple : `(nbytes, ancdata, msg_flags, address)`, where *nbytes* is the total number of bytes of non-ancillary data written into the buffers, and *ancdata*, *msg_flags* and *address* are the same as for `recvmsg()`.

Exemple :

```
>>> import socket
>>> s1, s2 = socket.socketpair()
>>> b1 = bytearray(b'----')
>>> b2 = bytearray(b'0123456789')
>>> b3 = bytearray(b'--------------')
>>> s1.send(b'Mary had a little lamb')
22
>>> s2.recvmsg_into([b1, memoryview(b2)[2:9], b3])
(22, [], 0, None)
>>> [b1, b2, b3]
[bytearray(b'Mary'), bytearray(b'01 had a 9'), bytearray(b'little lamb---')]
```

Availability : Unix.

Most Unix platforms.

Ajouté dans la version 3.3.

socket.**recvfrom_into**(*buffer*[, *nbytes*[, *flags*]])

Receive data from the socket, writing it into *buffer* instead of creating a new bytestring. The return value is a pair `(nbytes, address)` where *nbytes* is the number of bytes received and *address* is the address of the socket sending the data. See the Unix manual page `recv(2)` for the meaning of the optional argument *flags*; it defaults to zero. (The format of *address* depends on the address family --- see above.)

socket.**recv_into**(*buffer*[, *nbytes*[, *flags*]])

Receive up to *nbytes* bytes from the socket, storing the data into a buffer rather than creating a new bytestring. If *nbytes* is not specified (or 0), receive up to the size available in the given buffer. Returns the number of bytes received. See the Unix manual page `recv(2)` for the meaning of the optional argument *flags*; it defaults to zero.

socket.**send**(*bytes*[, *flags*])

Send data to the socket. The socket must be connected to a remote socket. The optional *flags* argument has the same meaning as for `recv()` above. Returns the number of bytes sent. Applications are responsible for checking that all data has been sent; if only some of the data was transmitted, the application needs to attempt delivery of the remaining data. For further information on this topic, consult the socket-howto.

Modifié dans la version 3.5 : Si l'appel système est interrompu et que le gestionnaire de signal ne lève aucune exception, la fonction réessaye l'appel système au lieu de lever une `InterruptedError` (voir la **PEP 475** à propos du raisonnement).

socket.**sendall**(*bytes*[, *flags*])

Send data to the socket. The socket must be connected to a remote socket. The optional *flags* argument has the same meaning as for `recv()` above. Unlike `send()`, this method continues to send data from *bytes* until either all data has been sent or an error occurs. `None` is returned on success. On error, an exception is raised, and there is no way to determine how much data, if any, was successfully sent.

Modifié dans la version 3.5 : The socket timeout is no longer reset each time data is sent successfully. The socket timeout is now the maximum total duration to send all data.

Modifié dans la version 3.5 : Si l'appel système est interrompu et que le gestionnaire de signal ne lève aucune exception, la fonction réessaye l'appel système au lieu de lever une `InterruptedError` (voir la **PEP 475** à propos du raisonnement).

socket.**sendto**(*bytes*, *address*)

socket.**sendto**(*bytes*, *flags*, *address*)

Send data to the socket. The socket should not be connected to a remote socket, since the destination socket is specified by *address*. The optional *flags* argument has the same meaning as for `recv()` above. Return the number of bytes sent. (The format of *address* depends on the address family --- see above.)

Raises an *auditing event* `socket.sendto` with arguments `self`, `address`.

Modifié dans la version 3.5 : Si l'appel système est interrompu et que le gestionnaire de signal ne lève aucune exception, la fonction réessaye l'appel système au lieu de lever une `InterruptedError` (voir la **PEP 475** à propos du raisonnement).

socket.**sendmsg**(*buffers*[, *ancdata*[, *flags*[, *address*]]])

Send normal and ancillary data to the socket, gathering the non-ancillary data from a series of buffers and concatenating it into a single message. The *buffers* argument specifies the non-ancillary data as an iterable of *bytes-like objects* (e.g. `bytes` objects); the operating system may set a limit (`sysconf()` value `SC_IOV_MAX`) on the number of buffers that can be used. The *ancdata* argument specifies the ancillary data (control messages) as an iterable of zero or more tuples `(cmsg_level, cmsg_type, cmsg_data)`, where *cmsg_level* and *cmsg_type* are integers specifying the protocol level and protocol-specific type respectively, and *cmsg_data* is a bytes-like object holding the associated data. Note that some systems (in particular, systems without `CMSG_SPACE()`) might support sending only one control message per call. The *flags* argument defaults to 0 and has the same meaning as for `send()`. If *address* is supplied and not `None`, it sets a destination address for the message. The return value is the number of bytes of non-ancillary data sent.

The following function sends the list of file descriptors *fds* over an `AF_UNIX` socket, on systems which support the `SCM_RIGHTS` mechanism. See also `recvmsg()`.

```
import socket, array

def send_fds(sock, msg, fds):
    return sock.sendmsg([msg], [(socket.SOL_SOCKET, socket.SCM_RIGHTS, array.
↪array("i", fds))])
```

Availability : Unix, not WASI.

Most Unix platforms.

Raises an *auditing event* `socket.sendmsg` with arguments `self`, `address`.

Ajouté dans la version 3.3.

Modifié dans la version 3.5 : Si l'appel système est interrompu et que le gestionnaire de signal ne lève aucune exception, la fonction réessaye l'appel système au lieu de lever une `InterruptedError` (voir la **PEP 475** à propos du raisonnement).

`socket.`**`sendmsg_afalg`**(*[msg,]*, op[, iv[, assoclen[, flags]]]*)

Specialized version of `sendmsg()` for `AF_ALG` socket. Set mode, IV, AEAD associated data length and flags for `AF_ALG` socket.

Availability : Linux >= 2.6.38.

Ajouté dans la version 3.6.

`socket.`**`sendfile`**(*file, offset=0, count=None*)

Send a file until EOF is reached by using high-performance `os.sendfile` and return the total number of bytes which were sent. *file* must be a regular file object opened in binary mode. If `os.sendfile` is not available (e.g. Windows) or *file* is not a regular file `send()` will be used instead. *offset* tells from where to start reading the file. If specified, *count* is the total number of bytes to transmit as opposed to sending the file until EOF is reached. File position is updated on return or also in case of error in which case `file.tell()` can be used to figure out the number of bytes which were sent. The socket must be of `SOCK_STREAM` type. Non-blocking sockets are not supported.

Ajouté dans la version 3.5.

`socket.`**`set_inheritable`**(*inheritable*)

Set the *inheritable flag* of the socket's file descriptor or socket's handle.

Ajouté dans la version 3.4.

`socket.`**`setblocking`**(*flag*)

Set blocking or non-blocking mode of the socket : if *flag* is false, the socket is set to non-blocking, else to blocking mode.

This method is a shorthand for certain `settimeout()` calls :
— `sock.setblocking(True)` is equivalent to `sock.settimeout(None)`
— `sock.setblocking(False)` is equivalent to `sock.settimeout(0.0)`
Modifié dans la version 3.7 : The method no longer applies `SOCK_NONBLOCK` flag on `socket.type`.

`socket.`**`settimeout`**(*value*)

Set a timeout on blocking socket operations. The *value* argument can be a nonnegative floating-point number expressing seconds, or `None`. If a non-zero value is given, subsequent socket operations will raise a `timeout` exception if the timeout period *value* has elapsed before the operation has completed. If zero is given, the socket is put in non-blocking mode. If `None` is given, the socket is put in blocking mode.

For further information, please consult the *notes on socket timeouts*.

Modifié dans la version 3.7 : The method no longer toggles `SOCK_NONBLOCK` flag on `socket.type`.

`socket.`**`setsockopt`**(*level, optname, value :* int)

`socket.`**`setsockopt`**(*level, optname, value : buffer*)

`socket.setsockopt(level, optname, None, optlen : int)`

Set the value of the given socket option (see the Unix manual page `setsockopt(2)`). The needed symbolic constants are defined in this module (SO_* etc. <socket-unix-constants>). The value can be an integer, `None` or a *bytes-like object* representing a buffer. In the later case it is up to the caller to ensure that the bytestring contains the proper bits (see the optional built-in module `struct` for a way to encode C structures as bytestrings). When *value* is set to `None`, *optlen* argument is required. It's equivalent to call `setsockopt()` C function with `optval=NULL` and `optlen=optlen`.

Modifié dans la version 3.5 : N'importe quel *bytes-like object* est maintenant accepté.

Modifié dans la version 3.6 : setsockopt(level, optname, None, optlen : int) form added.

Availability : not WASI.

`socket.shutdown(how)`

Shut down one or both halves of the connection. If *how* is `SHUT_RD`, further receives are disallowed. If *how* is `SHUT_WR`, further sends are disallowed. If *how* is `SHUT_RDWR`, further sends and receives are disallowed.

Availability : not WASI.

`socket.share(process_id)`

Duplicate a socket and prepare it for sharing with a target process. The target process must be provided with *process_id*. The resulting bytes object can then be passed to the target process using some form of interprocess communication and the socket can be recreated there using `fromshare()`. Once this method has been called, it is safe to close the socket since the operating system has already duplicated it for the target process.

Availability : Windows.

Ajouté dans la version 3.3.

Note that there are no methods `read()` or `write()` ; use `recv()` and `send()` without *flags* argument instead.

Socket objects also have these (read-only) attributes that correspond to the values given to the `socket` constructor.

`socket.family`

The socket family.

`socket.type`

The socket type.

`socket.proto`

The socket protocol.

19.2.4 Notes on socket timeouts

A socket object can be in one of three modes : blocking, non-blocking, or timeout. Sockets are by default always created in blocking mode, but this can be changed by calling `setdefaulttimeout()`.

- In *blocking mode*, operations block until complete or the system returns an error (such as connection timed out).
- In *non-blocking mode*, operations fail (with an error that is unfortunately system-dependent) if they cannot be completed immediately : functions from the `select` module can be used to know when and whether a socket is available for reading or writing.
- In *timeout mode*, operations fail if they cannot be completed within the timeout specified for the socket (they raise a `timeout` exception) or if the system returns an error.

Note

At the operating system level, sockets in *timeout mode* are internally set in non-blocking mode. Also, the blocking and timeout modes are shared between file descriptors and socket objects that refer to the same network endpoint. This implementation detail can have visible consequences if e.g. you decide to use the `fileno()` of a socket.

Timeouts and the `connect` method

The `connect()` operation is also subject to the timeout setting, and in general it is recommended to call `settimeout()` before calling `connect()` or pass a timeout parameter to `create_connection()`. However, the system network stack may also return a connection timeout error of its own regardless of any Python socket timeout setting.

Timeouts and the `accept` method

If `getdefaulttimeout()` is not `None`, sockets returned by the `accept()` method inherit that timeout. Otherwise, the behaviour depends on settings of the listening socket :

- if the listening socket is in *blocking mode* or in *timeout mode*, the socket returned by `accept()` is in *blocking mode* ;
- if the listening socket is in *non-blocking mode*, whether the socket returned by `accept()` is in blocking or non-blocking mode is operating system-dependent. If you want to ensure cross-platform behaviour, it is recommended you manually override this setting.

19.2.5 Exemple

Here are four minimal example programs using the TCP/IP protocol : a server that echoes all data that it receives back (servicing only one client), and a client using it. Note that a server must perform the sequence `socket()`, `bind()`, `listen()`, `accept()` (possibly repeating the `accept()` to service more than one client), while a client only needs the sequence `socket()`, `connect()`. Also note that the server does not `sendall()`/`recv()` on the socket it is listening on but on the new socket returned by `accept()`.

The first two examples support IPv4 only.

```
# Echo server program
import socket

HOST = ''                   # Symbolic name meaning all available interfaces
PORT = 50007                # Arbitrary non-privileged port
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen(1)
    conn, addr = s.accept()
    with conn:
        print('Connected by', addr)
        while True:
            data = conn.recv(1024)
            if not data: break
            conn.sendall(data)
```

```
# Echo client program
import socket

HOST = 'daring.cwi.nl'      # The remote host
PORT = 50007                # The same port as used by the server
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect((HOST, PORT))
    s.sendall(b'Hello, world')
    data = s.recv(1024)
print('Received', repr(data))
```

The next two examples are identical to the above two, but support both IPv4 and IPv6. The server side will listen to the first address family available (it should listen to both instead). On most of IPv6-ready systems, IPv6 will take precedence and the server may not accept IPv4 traffic. The client side will try to connect to all the addresses returned as a result of the name resolution, and sends traffic to the first one connected successfully.

```
# Echo server program
import socket
import sys

HOST = None               # Symbolic name meaning all available interfaces
PORT = 50007              # Arbitrary non-privileged port
s = None
for res in socket.getaddrinfo(HOST, PORT, socket.AF_UNSPEC,
                              socket.SOCK_STREAM, 0, socket.AI_PASSIVE):
    af, socktype, proto, canonname, sa = res
    try:
        s = socket.socket(af, socktype, proto)
    except OSError as msg:
        s = None
        continue
    try:
        s.bind(sa)
        s.listen(1)
    except OSError as msg:
        s.close()
        s = None
        continue
    break
if s is None:
    print('could not open socket')
    sys.exit(1)
conn, addr = s.accept()
with conn:
    print('Connected by', addr)
    while True:
        data = conn.recv(1024)
        if not data: break
        conn.send(data)
```

```
# Echo client program
import socket
import sys

HOST = 'daring.cwi.nl'    # The remote host
PORT = 50007              # The same port as used by the server
s = None
for res in socket.getaddrinfo(HOST, PORT, socket.AF_UNSPEC, socket.SOCK_STREAM):
    af, socktype, proto, canonname, sa = res
    try:
        s = socket.socket(af, socktype, proto)
    except OSError as msg:
        s = None
        continue
    try:
        s.connect(sa)
    except OSError as msg:
        s.close()
        s = None
        continue
    break
if s is None:
```

(suite sur la page suivante)

(suite de la page précédente)

```
    print('could not open socket')
    sys.exit(1)
with s:
    s.sendall(b'Hello, world')
    data = s.recv(1024)
print('Received', repr(data))
```

The next example shows how to write a very simple network sniffer with raw sockets on Windows. The example requires administrator privileges to modify the interface :

```
import socket

# the public network interface
HOST = socket.gethostbyname(socket.gethostname())

# create a raw socket and bind it to the public interface
s = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_IP)
s.bind((HOST, 0))

# Include IP headers
s.setsockopt(socket.IPPROTO_IP, socket.IP_HDRINCL, 1)

# receive all packets
s.ioctl(socket.SIO_RCVALL, socket.RCVALL_ON)

# receive a packet
print(s.recvfrom(65565))

# disabled promiscuous mode
s.ioctl(socket.SIO_RCVALL, socket.RCVALL_OFF)
```

The next example shows how to use the socket interface to communicate to a CAN network using the raw socket protocol. To use CAN with the broadcast manager protocol instead, open a socket with :

```
socket.socket(socket.AF_CAN, socket.SOCK_DGRAM, socket.CAN_BCM)
```

After binding (`CAN_RAW`) or connecting (`CAN_BCM`) the socket, you can use the `socket.send()` and `socket.recv()` operations (and their counterparts) on the socket object as usual.

This last example might require special privileges :

```
import socket
import struct


# CAN frame packing/unpacking (see 'struct can_frame' in <linux/can.h>)

can_frame_fmt = "=IB3x8s"
can_frame_size = struct.calcsize(can_frame_fmt)

def build_can_frame(can_id, data):
    can_dlc = len(data)
    data = data.ljust(8, b'\x00')
    return struct.pack(can_frame_fmt, can_id, can_dlc, data)

def dissect_can_frame(frame):
    can_id, can_dlc, data = struct.unpack(can_frame_fmt, frame)
```

(suite sur la page suivante)

(suite de la page précédente)

```
    return (can_id, can_dlc, data[:can_dlc])


# create a raw socket and bind it to the 'vcan0' interface
s = socket.socket(socket.AF_CAN, socket.SOCK_RAW, socket.CAN_RAW)
s.bind(('vcan0',))

while True:
    cf, addr = s.recvfrom(can_frame_size)

    print('Received: can_id=%x, can_dlc=%x, data=%s' % dissect_can_frame(cf))

    try:
        s.send(cf)
    except OSError:
        print('Error sending CAN frame')

    try:
        s.send(build_can_frame(0x01, b'\x01\x02\x03'))
    except OSError:
        print('Error sending CAN frame')
```

Running an example several times with too small delay between executions, could lead to this error :

```
OSError: [Errno 98] Address already in use
```

This is because the previous execution has left the socket in a `TIME_WAIT` state, and can't be immediately reused.

There is a *socket* flag to set, in order to prevent this, `socket.SO_REUSEADDR` :

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
s.bind((HOST, PORT))
```

the `SO_REUSEADDR` flag tells the kernel to reuse a local socket in `TIME_WAIT` state, without waiting for its natural timeout to expire.

Voir aussi

For an introduction to socket programming (in C), see the following papers :

— *An Introductory 4.3BSD Interprocess Communication Tutorial*, by Stuart Sechrest

— *An Advanced 4.3BSD Interprocess Communication Tutorial*, by Samuel J. Leffler et al,

both in the UNIX Programmer's Manual, Supplementary Documents 1 (sections PS1 :7 and PS1 :8). The platform-specific reference material for the various socket-related system calls are also a valuable source of information on the details of socket semantics. For Unix, refer to the manual pages ; for Windows, see the WinSock (or Winsock 2) specification. For IPv6-ready APIs, readers may want to refer to **RFC 3493** titled Basic Socket Interface Extensions for IPv6.

19.3 `ssl` --- TLS/SSL wrapper for socket objects

Code source : Lib/ssl.py

This module provides access to Transport Layer Security (often known as "Secure Sockets Layer") encryption and peer authentication facilities for network sockets, both client-side and server-side. This module uses the OpenSSL

library. It is available on all modern Unix systems, Windows, macOS, and probably additional platforms, as long as OpenSSL is installed on that platform.

> **Note**
>
> Some behavior may be platform dependent, since calls are made to the operating system socket APIs. The installed version of OpenSSL may also cause variations in behavior. For example, TLSv1.3 comes with OpenSSL version 1.1.1.

> **Avertissement**
>
> N'utilisez pas ce module sans lire *Security considerations*. Cela pourrait créer un faux sentiment de sécurité, car les paramètres par défaut du module *ssl* ne sont pas nécessairement appropriés pour votre application.

Availability : not WASI.

This module does not work or is not available on WebAssembly. See *Plateformes WebAssembly* for more information.

Cette section documente les objets et les fonctions du module `ssl`. Pour des informations plus générales sur TLS, SSL et les certificats, le lecteur est prié de se référer aux documents de la section « Voir Aussi » au bas de cette page.

This module provides a class, `ssl.SSLSocket`, which is derived from the `socket.socket` type, and provides a socket-like wrapper that also encrypts and decrypts the data going over the socket with SSL. It supports additional methods such as `getpeercert()`, which retrieves the certificate of the other side of the connection, `cipher()`, which retrieves the cipher being used for the secure connection or `get_verified_chain()`, `get_unverified_chain()` which retrieves certificate chain.

Pour les applications plus sophistiquées, la classe `ssl.SSLContext` facilite la gestion des paramètres et des certificats, qui peuvent ensuite être hérités par les connecteurs SSL créés via la méthode `SSLContext.wrap_socket()`.

Modifié dans la version 3.5.3 : Mise à jour pour prendre en charge la liaison avec OpenSSL 1.1.0

Modifié dans la version 3.6 : OpenSSL 0.9.8, 1.0.0 et 1.0.1 sont obsolètes et ne sont plus prises en charge. Dans l'avenir, le module *ssl* nécessitera au minimum OpenSSL 1.0.2 ou 1.1.0.

Modifié dans la version 3.10 : **PEP 644** has been implemented. The ssl module requires OpenSSL 1.1.1 or newer.

Use of deprecated constants and functions result in deprecation warnings.

19.3.1 Fonctions, constantes et exceptions

Création de connecteurs

Instances of `SSLSocket` must be created using the `SSLContext.wrap_socket()` method. The helper function `create_default_context()` returns a new context with secure default settings.

Exemple de connecteur client avec contexte par défaut et double pile IPv4/IPv6 :

```
import socket
import ssl

hostname = 'www.python.org'
context = ssl.create_default_context()

with socket.create_connection((hostname, 443)) as sock:
    with context.wrap_socket(sock, server_hostname=hostname) as ssock:
        print(ssock.version())
```

Exemple de connecteur client avec contexte personnalisé et IPv4 :

```
hostname = 'www.python.org'
# PROTOCOL_TLS_CLIENT requires valid cert chain and hostname
context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
context.load_verify_locations('path/to/cabundle.pem')

with socket.socket(socket.AF_INET, socket.SOCK_STREAM, 0) as sock:
    with context.wrap_socket(sock, server_hostname=hostname) as ssock:
        print(ssock.version())
```

Exemple de connecteur serveur à l'écoute sur IPv4 *localhost* :

```
context = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)
context.load_cert_chain('/path/to/certchain.pem', '/path/to/private.key')

with socket.socket(socket.AF_INET, socket.SOCK_STREAM, 0) as sock:
    sock.bind(('127.0.0.1', 8443))
    sock.listen(5)
    with context.wrap_socket(sock, server_side=True) as ssock:
        conn, addr = ssock.accept()
        ...
```

Création de contexte

Une fonction utilitaire permettant de créer facilement des objets `SSLContext` pour des usages classiques.

`ssl.`**`create_default_context`**(*purpose=Purpose.SERVER_AUTH*, *cafile=None*, *capath=None*, *cadata=None*)

Renvoie un nouvel objet `SSLContext`. Le paramètre *purpose* permet de choisir parmi un ensemble de paramètres par défaut en fonction de l'usage souhaité. Les paramètres sont choisis par le module `ssl` et représentent généralement un niveau de sécurité supérieur à celui utilisé lorsque vous appelez directement le constructeur `SSLContext`.

cafile, *capath*, *cadata* représentent des certificats d'autorité de certification facultatifs approuvés pour la vérification de certificats, comme dans `SSLContext.load_verify_locations()`. Si les trois sont à `None`, cette fonction peut choisir de faire confiance aux certificats d'autorité de certification par défaut du système.

The settings are : `PROTOCOL_TLS_CLIENT` or `PROTOCOL_TLS_SERVER`, `OP_NO_SSLv2`, and `OP_NO_SSLv3` with high encryption cipher suites without RC4 and without unauthenticated cipher suites. Passing `SERVER_AUTH` as *purpose* sets `verify_mode` to `CERT_REQUIRED` and either loads CA certificates (when at least one of *cafile*, *capath* or *cadata* is given) or uses `SSLContext.load_default_certs()` to load default CA certificates.

Lorsque `keylog_filename` est pris en charge et que la variable d'environnement `SSLKEYLOGFILE` est définie, `create_default_context()` active la journalisation des clés.

The default settings for this context include `VERIFY_X509_PARTIAL_CHAIN` and `VERIFY_X509_STRICT`. These make the underlying OpenSSL implementation behave more like a conforming implementation of **RFC 5280**, in exchange for a small amount of incompatibility with older X.509 certificates.

> **Note**
>
> Le protocole, les options, l'algorithme de chiffrement et d'autres paramètres peuvent changer pour des valeurs plus restrictives à tout moment sans avertissement préalable. Les valeurs représentent un juste équilibre entre compatibilité et sécurité.
>
> Si votre application nécessite des paramètres spécifiques, vous devez créer une classe `SSLContext` et appliquer les paramètres vous-même.

> **Note**
>
> Si vous constatez que, lorsque certains clients ou serveurs plus anciens tentent de se connecter avec une classe `SSLContext` créée par cette fonction, une erreur indiquant « *Protocol or cipher suite mismatch* » (« Non concordance de protocole ou d'algorithme de chiffrement ») est détectée, il se peut qu'ils ne prennent en charge que SSL 3.0 que cette fonction exclut en utilisant `OP_NO_SSLv3`. SSL3.0 est notoirement considéré comme totalement déficient. Si vous souhaitez toujours continuer à utiliser cette fonction tout en autorisant les connexions SSL 3.0, vous pouvez les réactiver à l'aide de :
>
> ```
> ctx = ssl.create_default_context(Purpose.CLIENT_AUTH)
> ctx.options &= ~ssl.OP_NO_SSLv3
> ```

> **Note**
>
> This context enables `VERIFY_X509_STRICT` by default, which may reject pre-**RFC 5280** or malformed certificates that the underlying OpenSSL implementation otherwise would accept. While disabling this is not recommended, you can do so using :
>
> ```
> ctx = ssl.create_default_context()
> ctx.verify_flags &= ~ssl.VERIFY_X509_STRICT
> ```

Ajouté dans la version 3.4.

Modifié dans la version 3.4.4 : RC4 a été supprimé de la liste des algorithmes de chiffrement par défaut.

Modifié dans la version 3.6 : *ChaCha20/Poly1305* a été ajouté à la liste des algorithmes de chiffrement par défaut.

3DES a été supprimé de la liste des algorithmes de chiffrement par défaut.

Modifié dans la version 3.8 : La prise en charge de la journalisation des clés dans `SSLKEYLOGFILE` a été ajoutée.

Modifié dans la version 3.10 : The context now uses `PROTOCOL_TLS_CLIENT` or `PROTOCOL_TLS_SERVER` protocol instead of generic `PROTOCOL_TLS`.

Modifié dans la version 3.13 : The context now uses `VERIFY_X509_PARTIAL_CHAIN` and `VERIFY_X509_STRICT` in its default verify flags.

Exceptions

exception ssl.**SSLError**

Levée pour signaler une erreur de l'implémentation SSL sous-jacente (actuellement fournie par la bibliothèque OpenSSL). Cela signifie qu'un problème est apparu dans la couche d'authentification et de chiffrement de niveau supérieur qui s'appuie sur la connexion réseau sous-jacente. Cette erreur est un sous-type de `OSError`. Le code d'erreur et le message des instances de `SSLError` sont fournis par la bibliothèque OpenSSL.

Modifié dans la version 3.3 : `SSLError` était un sous-type de `socket.error`.

library

Une chaîne de caractères mnémonique désignant le sous-module OpenSSL dans lequel l'erreur s'est produite, telle que `SSL`, `PEM` ou `X509`. L'étendue des valeurs possibles dépend de la version d'OpenSSL.

Ajouté dans la version 3.3.

reason

A string mnemonic designating the reason this error occurred, for example `CERTIFICATE_VERIFY_FAILED`. The range of possible values depends on the OpenSSL version.

Ajouté dans la version 3.3.

exception ssl.**SSLZeroReturnError**

A subclass of *SSLError* raised when trying to read or write and the SSL connection has been closed cleanly. Note that this doesn't mean that the underlying transport (read TCP) has been closed.

Ajouté dans la version 3.3.

exception ssl.**SSLWantReadError**

Sous-classe de *SSLError* levée par un connecteur *SSL non bloquant* lors d'une tentative de lecture ou d'écriture de données, alors que davantage de données doivent être reçues sur la couche TCP sous-jacente avant que la demande puisse être satisfaite.

Ajouté dans la version 3.3.

exception ssl.**SSLWantWriteError**

A subclass of *SSLError* raised by a *non-blocking SSL socket* when trying to read or write data, but more data needs to be sent on the underlying TCP transport before the request can be fulfilled.

Ajouté dans la version 3.3.

exception ssl.**SSLSyscallError**

A subclass of *SSLError* raised when a system error was encountered while trying to fulfill an operation on a SSL socket. Unfortunately, there is no easy way to inspect the original errno number.

Ajouté dans la version 3.3.

exception ssl.**SSLEOFError**

A subclass of *SSLError* raised when the SSL connection has been terminated abruptly. Generally, you shouldn't try to reuse the underlying transport when this error is encountered.

Ajouté dans la version 3.3.

exception ssl.**SSLCertVerificationError**

A subclass of *SSLError* raised when certificate validation has failed.

Ajouté dans la version 3.7.

verify_code

A numeric error number that denotes the verification error.

verify_message

A human readable string of the verification error.

exception ssl.**CertificateError**

An alias for *SSLCertVerificationError*.

Modifié dans la version 3.7 : The exception is now an alias for *SSLCertVerificationError*.

Random generation

ssl.**RAND_bytes**(*num*)

Return *num* cryptographically strong pseudo-random bytes. Raises an *SSLError* if the PRNG has not been seeded with enough data or if the operation is not supported by the current RAND method. *RAND_status()* can be used to check the status of the PRNG and *RAND_add()* can be used to seed the PRNG.

For almost all applications *os.urandom()* is preferable.

Read the Wikipedia article, Cryptographically secure pseudorandom number generator (CSPRNG), to get the requirements of a cryptographically strong generator.

Ajouté dans la version 3.3.

ssl.**RAND_status**()

Return `True` if the SSL pseudo-random number generator has been seeded with 'enough' randomness, and `False` otherwise. You can use `ssl.RAND_egd()` and *ssl.RAND_add()* to increase the randomness of the pseudo-random number generator.

ssl.**RAND_add**(*bytes*, *entropy*)

Mix the given *bytes* into the SSL pseudo-random number generator. The parameter *entropy* (a float) is a lower bound on the entropy contained in string (so you can always use `0.0`). See **RFC 1750** for more information on sources of entropy.

Modifié dans la version 3.5 : N'importe quel *bytes-like object* est maintenant accepté.

Certificate handling

ssl.**cert_time_to_seconds**(*cert_time*)

Return the time in seconds since the Epoch, given the `cert_time` string representing the "notBefore" or "notAfter" date from a certificate in `"%b %d %H:%M:%S %Y %Z"` strptime format (C locale).

Here's an example :

```
>>> import ssl
>>> timestamp = ssl.cert_time_to_seconds("Jan  5 09:34:43 2018 GMT")
>>> timestamp
1515144883
>>> from datetime import datetime
>>> print(datetime.utcfromtimestamp(timestamp))
2018-01-05 09:34:43
```

"notBefore" or "notAfter" dates must use GMT (**RFC 5280**).

Modifié dans la version 3.5 : Interpret the input time as a time in UTC as specified by 'GMT' timezone in the input string. Local timezone was used previously. Return an integer (no fractions of a second in the input format)

ssl.**get_server_certificate**(*addr*, *ssl_version=PROTOCOL_TLS_CLIENT*, *ca_certs=None*[, *timeout*])

Given the address `addr` of an SSL-protected server, as a (*hostname*, *port-number*) pair, fetches the server's certificate, and returns it as a PEM-encoded string. If `ssl_version` is specified, uses that version of the SSL protocol to attempt to connect to the server. If *ca_certs* is specified, it should be a file containing a list of root certificates, the same format as used for the *cafile* parameter in `SSLContext.load_verify_locations()`. The call will attempt to validate the server certificate against that set of root certificates, and will fail if the validation attempt fails. A timeout can be specified with the `timeout` parameter.

Modifié dans la version 3.3 : This function is now IPv6-compatible.

Modifié dans la version 3.5 : The default *ssl_version* is changed from `PROTOCOL_SSLv3` to `PROTOCOL_TLS` for maximum compatibility with modern servers.

Modifié dans la version 3.10 : The *timeout* parameter was added.

ssl.**DER_cert_to_PEM_cert**(*DER_cert_bytes*)

Given a certificate as a DER-encoded blob of bytes, returns a PEM-encoded string version of the same certificate.

ssl.**PEM_cert_to_DER_cert**(*PEM_cert_string*)

Given a certificate as an ASCII PEM string, returns a DER-encoded sequence of bytes for that same certificate.

ssl.**get_default_verify_paths**()

Returns a named tuple with paths to OpenSSL's default cafile and capath. The paths are the same as used by `SSLContext.set_default_verify_paths()`. The return value is a *named tuple* `DefaultVerifyPaths` :

— `cafile` - resolved path to cafile or `None` if the file doesn't exist,
— `capath` - resolved path to capath or `None` if the directory doesn't exist,
— `openssl_cafile_env` - OpenSSL's environment key that points to a cafile,
— `openssl_cafile` - hard coded path to a cafile,
— `openssl_capath_env` - OpenSSL's environment key that points to a capath,
— `openssl_capath` - hard coded path to a capath directory

Ajouté dans la version 3.4.

ssl.**enum_certificates**(*store_name*)

Retrieve certificates from Windows' system cert store. *store_name* may be one of CA, ROOT or MY. Windows may provide additional cert stores, too.

The function returns a list of (cert_bytes, encoding_type, trust) tuples. The encoding_type specifies the encoding of cert_bytes. It is either x509_asn for X.509 ASN.1 data or pkcs_7_asn for PKCS#7 ASN.1 data. Trust specifies the purpose of the certificate as a set of OIDS or exactly True if the certificate is trustworthy for all purposes.

Exemple :

```
>>> ssl.enum_certificates("CA")
[(b'data...', 'x509_asn', {'1.3.6.1.5.5.7.3.1', '1.3.6.1.5.5.7.3.2'}),
 (b'data...', 'x509_asn', True)]
```

Availability : Windows.

Ajouté dans la version 3.4.

ssl.**enum_crls**(*store_name*)

Retrieve CRLs from Windows' system cert store. *store_name* may be one of CA, ROOT or MY. Windows may provide additional cert stores, too.

The function returns a list of (cert_bytes, encoding_type, trust) tuples. The encoding_type specifies the encoding of cert_bytes. It is either x509_asn for X.509 ASN.1 data or pkcs_7_asn for PKCS#7 ASN.1 data.

Availability : Windows.

Ajouté dans la version 3.4.

Constantes

All constants are now *enum.IntEnum* or *enum.IntFlag* collections.

Ajouté dans la version 3.6.

ssl.**CERT_NONE**

Possible value for *SSLContext.verify_mode*. Except for *PROTOCOL_TLS_CLIENT*, it is the default mode. With client-side sockets, just about any cert is accepted. Validation errors, such as untrusted or expired cert, are ignored and do not abort the TLS/SSL handshake.

In server mode, no certificate is requested from the client, so the client does not send any for client cert authentication.

See the discussion of *Security considerations* below.

ssl.**CERT_OPTIONAL**

Possible value for *SSLContext.verify_mode*. In client mode, *CERT_OPTIONAL* has the same meaning as *CERT_REQUIRED*. It is recommended to use *CERT_REQUIRED* for client-side sockets instead.

In server mode, a client certificate request is sent to the client. The client may either ignore the request or send a certificate in order perform TLS client cert authentication. If the client chooses to send a certificate, it is verified. Any verification error immediately aborts the TLS handshake.

Use of this setting requires a valid set of CA certificates to be passed to *SSLContext.load_verify_locations()*.

ssl.**CERT_REQUIRED**

Possible value for *SSLContext.verify_mode*. In this mode, certificates are required from the other side of the socket connection ; an *SSLError* will be raised if no certificate is provided, or if its validation fails. This mode is **not** sufficient to verify a certificate in client mode as it does not match hostnames. *check_hostname* must be enabled as well to verify the authenticity of a cert. *PROTOCOL_TLS_CLIENT* uses *CERT_REQUIRED* and enables *check_hostname* by default.

With server socket, this mode provides mandatory TLS client cert authentication. A client certificate request is sent to the client and the client must provide a valid and trusted certificate.

Use of this setting requires a valid set of CA certificates to be passed to *SSLContext.load_verify_locations()*.

class ssl.**VerifyMode**

enum.IntEnum collection of CERT_* constants.

Ajouté dans la version 3.6.

ssl.**VERIFY_DEFAULT**

Possible value for *SSLContext.verify_flags*. In this mode, certificate revocation lists (CRLs) are not checked. By default OpenSSL does neither require nor verify CRLs.

Ajouté dans la version 3.4.

ssl.**VERIFY_CRL_CHECK_LEAF**

Possible value for *SSLContext.verify_flags*. In this mode, only the peer cert is checked but none of the intermediate CA certificates. The mode requires a valid CRL that is signed by the peer cert's issuer (its direct ancestor CA). If no proper CRL has been loaded with *SSLContext.load_verify_locations*, validation will fail.

Ajouté dans la version 3.4.

ssl.**VERIFY_CRL_CHECK_CHAIN**

Possible value for *SSLContext.verify_flags*. In this mode, CRLs of all certificates in the peer cert chain are checked.

Ajouté dans la version 3.4.

ssl.**VERIFY_X509_STRICT**

Possible value for *SSLContext.verify_flags* to disable workarounds for broken X.509 certificates.

Ajouté dans la version 3.4.

ssl.**VERIFY_ALLOW_PROXY_CERTS**

Possible value for *SSLContext.verify_flags* to enables proxy certificate verification.

Ajouté dans la version 3.10.

ssl.**VERIFY_X509_TRUSTED_FIRST**

Possible value for *SSLContext.verify_flags*. It instructs OpenSSL to prefer trusted certificates when building the trust chain to validate a certificate. This flag is enabled by default.

Ajouté dans la version 3.4.4.

ssl.**VERIFY_X509_PARTIAL_CHAIN**

Possible value for *SSLContext.verify_flags*. It instructs OpenSSL to accept intermediate CAs in the trust store to be treated as trust-anchors, in the same way as the self-signed root CA certificates. This makes it possible to trust certificates issued by an intermediate CA without having to trust its ancestor root CA.

Ajouté dans la version 3.10.

class ssl.**VerifyFlags**

enum.IntFlag collection of VERIFY_* constants.

Ajouté dans la version 3.6.

ssl.**PROTOCOL_TLS**

Selects the highest protocol version that both the client and server support. Despite the name, this option can select both "SSL" and "TLS" protocols.

Ajouté dans la version 3.6.

Obsolète depuis la version 3.10 : TLS clients and servers require different default settings for secure communication. The generic TLS protocol constant is deprecated in favor of *PROTOCOL_TLS_CLIENT* and *PROTOCOL_TLS_SERVER*.

ssl.**PROTOCOL_TLS_CLIENT**

Auto-negotiate the highest protocol version that both the client and server support, and configure the context client-side connections. The protocol enables *CERT_REQUIRED* and *check_hostname* by default.

Ajouté dans la version 3.6.

ssl.**PROTOCOL_TLS_SERVER**

Auto-negotiate the highest protocol version that both the client and server support, and configure the context server-side connections.

Ajouté dans la version 3.6.

ssl.**PROTOCOL_SSLv23**

Alias for *PROTOCOL_TLS*.

Obsolète depuis la version 3.6 : Use *PROTOCOL_TLS* instead.

ssl.**PROTOCOL_SSLv3**

Selects SSL version 3 as the channel encryption protocol.

This protocol is not available if OpenSSL is compiled with the `no-ssl3` option.

> **Avertissement**
>
> SSL version 3 is insecure. Its use is highly discouraged.

Obsolète depuis la version 3.6 : OpenSSL has deprecated all version specific protocols. Use the default protocol *PROTOCOL_TLS_SERVER* or *PROTOCOL_TLS_CLIENT* with *SSLContext.minimum_version* and *SSLContext.maximum_version* instead.

ssl.**PROTOCOL_TLSv1**

Selects TLS version 1.0 as the channel encryption protocol.

Obsolète depuis la version 3.6 : OpenSSL has deprecated all version specific protocols.

ssl.**PROTOCOL_TLSv1_1**

Selects TLS version 1.1 as the channel encryption protocol. Available only with openssl version 1.0.1+.

Ajouté dans la version 3.4.

Obsolète depuis la version 3.6 : OpenSSL has deprecated all version specific protocols.

ssl.**PROTOCOL_TLSv1_2**

Selects TLS version 1.2 as the channel encryption protocol. Available only with openssl version 1.0.1+.

Ajouté dans la version 3.4.

Obsolète depuis la version 3.6 : OpenSSL has deprecated all version specific protocols.

ssl.**OP_ALL**

Enables workarounds for various bugs present in other SSL implementations. This option is set by default. It does not necessarily set the same flags as OpenSSL's `SSL_OP_ALL` constant.

Ajouté dans la version 3.2.

ssl.**OP_NO_SSLv2**

Prevents an SSLv2 connection. This option is only applicable in conjunction with *PROTOCOL_TLS*. It prevents the peers from choosing SSLv2 as the protocol version.

Ajouté dans la version 3.2.

Obsolète depuis la version 3.6 : SSLv2 is deprecated

ssl.**OP_NO_SSLv3**

Prevents an SSLv3 connection. This option is only applicable in conjunction with *PROTOCOL_TLS*. It prevents the peers from choosing SSLv3 as the protocol version.

Ajouté dans la version 3.2.

Obsolète depuis la version 3.6 : SSLv3 is deprecated

ssl.**OP_NO_TLSv1**

Prevents a TLSv1 connection. This option is only applicable in conjunction with *PROTOCOL_TLS*. It prevents the peers from choosing TLSv1 as the protocol version.

Ajouté dans la version 3.2.

Obsolète depuis la version 3.7 : The option is deprecated since OpenSSL 1.1.0, use the new *SSLContext.minimum_version* and *SSLContext.maximum_version* instead.

ssl.**OP_NO_TLSv1_1**

Prevents a TLSv1.1 connection. This option is only applicable in conjunction with *PROTOCOL_TLS*. It prevents the peers from choosing TLSv1.1 as the protocol version. Available only with openssl version 1.0.1+.

Ajouté dans la version 3.4.

Obsolète depuis la version 3.7 : The option is deprecated since OpenSSL 1.1.0.

ssl.**OP_NO_TLSv1_2**

Prevents a TLSv1.2 connection. This option is only applicable in conjunction with *PROTOCOL_TLS*. It prevents the peers from choosing TLSv1.2 as the protocol version. Available only with openssl version 1.0.1+.

Ajouté dans la version 3.4.

Obsolète depuis la version 3.7 : The option is deprecated since OpenSSL 1.1.0.

ssl.**OP_NO_TLSv1_3**

Prevents a TLSv1.3 connection. This option is only applicable in conjunction with *PROTOCOL_TLS*. It prevents the peers from choosing TLSv1.3 as the protocol version. TLS 1.3 is available with OpenSSL 1.1.1 or later. When Python has been compiled against an older version of OpenSSL, the flag defaults to *0*.

Ajouté dans la version 3.6.3.

Obsolète depuis la version 3.7 : The option is deprecated since OpenSSL 1.1.0. It was added to 2.7.15 and 3.6.3 for backwards compatibility with OpenSSL 1.0.2.

ssl.**OP_NO_RENEGOTIATION**

Disable all renegotiation in TLSv1.2 and earlier. Do not send HelloRequest messages, and ignore renegotiation requests via ClientHello.

This option is only available with OpenSSL 1.1.0h and later.

Ajouté dans la version 3.7.

ssl.**OP_CIPHER_SERVER_PREFERENCE**

Use the server's cipher ordering preference, rather than the client's. This option has no effect on client sockets and SSLv2 server sockets.

Ajouté dans la version 3.3.

ssl.**OP_SINGLE_DH_USE**

Prevents reuse of the same DH key for distinct SSL sessions. This improves forward secrecy but requires more computational resources. This option only applies to server sockets.

Ajouté dans la version 3.3.

ssl.**OP_SINGLE_ECDH_USE**

Prevents reuse of the same ECDH key for distinct SSL sessions. This improves forward secrecy but requires more computational resources. This option only applies to server sockets.

Ajouté dans la version 3.3.

ssl.**OP_ENABLE_MIDDLEBOX_COMPAT**

Send dummy Change Cipher Spec (CCS) messages in TLS 1.3 handshake to make a TLS 1.3 connection look more like a TLS 1.2 connection.

This option is only available with OpenSSL 1.1.1 and later.

Ajouté dans la version 3.8.

`ssl.OP_NO_COMPRESSION`

Disable compression on the SSL channel. This is useful if the application protocol supports its own compression scheme.

Ajouté dans la version 3.3.

class `ssl.Options`

`enum.IntFlag` collection of OP_* constants.

`ssl.OP_NO_TICKET`

Prevent client side from requesting a session ticket.

Ajouté dans la version 3.6.

`ssl.OP_IGNORE_UNEXPECTED_EOF`

Ignore unexpected shutdown of TLS connections.

This option is only available with OpenSSL 3.0.0 and later.

Ajouté dans la version 3.10.

`ssl.OP_ENABLE_KTLS`

Enable the use of the kernel TLS. To benefit from the feature, OpenSSL must have been compiled with support for it, and the negotiated cipher suites and extensions must be supported by it (a list of supported ones may vary by platform and kernel version).

Note that with enabled kernel TLS some cryptographic operations are performed by the kernel directly and not via any available OpenSSL Providers. This might be undesirable if, for example, the application requires all cryptographic operations to be performed by the FIPS provider.

This option is only available with OpenSSL 3.0.0 and later.

Ajouté dans la version 3.12.

`ssl.OP_LEGACY_SERVER_CONNECT`

Allow legacy insecure renegotiation between OpenSSL and unpatched servers only.

Ajouté dans la version 3.12.

`ssl.HAS_ALPN`

Whether the OpenSSL library has built-in support for the *Application-Layer Protocol Negotiation* TLS extension as described in **RFC 7301**.

Ajouté dans la version 3.5.

`ssl.HAS_NEVER_CHECK_COMMON_NAME`

Whether the OpenSSL library has built-in support not checking subject common name and `SSLContext.hostname_checks_common_name` is writeable.

Ajouté dans la version 3.7.

`ssl.HAS_ECDH`

Whether the OpenSSL library has built-in support for the Elliptic Curve-based Diffie-Hellman key exchange. This should be true unless the feature was explicitly disabled by the distributor.

Ajouté dans la version 3.3.

`ssl.HAS_SNI`

Whether the OpenSSL library has built-in support for the *Server Name Indication* extension (as defined in **RFC 6066**).

Ajouté dans la version 3.2.

ssl.**HAS_NPN**

Whether the OpenSSL library has built-in support for the *Next Protocol Negotiation* as described in the Application Layer Protocol Negotiation. When true, you can use the `SSLContext.set_npn_protocols()` method to advertise which protocols you want to support.

Ajouté dans la version 3.3.

ssl.**HAS_SSLv2**

Whether the OpenSSL library has built-in support for the SSL 2.0 protocol.

Ajouté dans la version 3.7.

ssl.**HAS_SSLv3**

Whether the OpenSSL library has built-in support for the SSL 3.0 protocol.

Ajouté dans la version 3.7.

ssl.**HAS_TLSv1**

Whether the OpenSSL library has built-in support for the TLS 1.0 protocol.

Ajouté dans la version 3.7.

ssl.**HAS_TLSv1_1**

Whether the OpenSSL library has built-in support for the TLS 1.1 protocol.

Ajouté dans la version 3.7.

ssl.**HAS_TLSv1_2**

Whether the OpenSSL library has built-in support for the TLS 1.2 protocol.

Ajouté dans la version 3.7.

ssl.**HAS_TLSv1_3**

Whether the OpenSSL library has built-in support for the TLS 1.3 protocol.

Ajouté dans la version 3.7.

ssl.**HAS_PSK**

Whether the OpenSSL library has built-in support for TLS-PSK.

Ajouté dans la version 3.13.

ssl.**HAS_PHA**

Whether the OpenSSL library has built-in support for TLS-PHA.

Ajouté dans la version 3.14.

ssl.**CHANNEL_BINDING_TYPES**

List of supported TLS channel binding types. Strings in this list can be used as arguments to `SSLSocket.get_channel_binding()`.

Ajouté dans la version 3.3.

ssl.**OPENSSL_VERSION**

The version string of the OpenSSL library loaded by the interpreter :

```
>>> ssl.OPENSSL_VERSION
'OpenSSL 1.0.2k  26 Jan 2017'
```

Ajouté dans la version 3.2.

ssl.**OPENSSL_VERSION_INFO**

A tuple of five integers representing version information about the OpenSSL library :

```
>>> ssl.OPENSSL_VERSION_INFO
(1, 0, 2, 11, 15)
```

Ajouté dans la version 3.2.

ssl.**OPENSSL_VERSION_NUMBER**

The raw version number of the OpenSSL library, as a single integer :

```
>>> ssl.OPENSSL_VERSION_NUMBER
268443839
>>> hex(ssl.OPENSSL_VERSION_NUMBER)
'0x100020bf'
```

Ajouté dans la version 3.2.

ssl.**ALERT_DESCRIPTION_HANDSHAKE_FAILURE**

ssl.**ALERT_DESCRIPTION_INTERNAL_ERROR**

ALERT_DESCRIPTION_*

Alert Descriptions from **RFC 5246** and others. The IANA TLS Alert Registry contains this list and references to the RFCs where their meaning is defined.

Used as the return value of the callback function in *SSLContext.set_servername_callback()*.

Ajouté dans la version 3.4.

class ssl.**AlertDescription**

enum.IntEnum collection of ALERT_DESCRIPTION_* constants.

Ajouté dans la version 3.6.

Purpose.**SERVER_AUTH**

Option for *create_default_context()* and *SSLContext.load_default_certs()*. This value indicates that the context may be used to authenticate web servers (therefore, it will be used to create client-side sockets).

Ajouté dans la version 3.4.

Purpose.**CLIENT_AUTH**

Option for *create_default_context()* and *SSLContext.load_default_certs()*. This value indicates that the context may be used to authenticate web clients (therefore, it will be used to create server-side sockets).

Ajouté dans la version 3.4.

class ssl.**SSLErrorNumber**

enum.IntEnum collection of SSL_ERROR_* constants.

Ajouté dans la version 3.6.

class ssl.**TLSVersion**

enum.IntEnum collection of SSL and TLS versions for *SSLContext.maximum_version* and *SSLContext.minimum_version*.

Ajouté dans la version 3.7.

TLSVersion.**MINIMUM_SUPPORTED**

TLSVersion.**MAXIMUM_SUPPORTED**

The minimum or maximum supported SSL or TLS version. These are magic constants. Their values don't reflect the lowest and highest available TLS/SSL versions.

TLSVersion.**SSLv3**

TLSVersion.**TLSv1**

TLSVersion.**TLSv1_1**

`TLSVersion.TLSv1_2`

`TLSVersion.TLSv1_3`

SSL 3.0 to TLS 1.3.

Obsolète depuis la version 3.10 : All `TLSVersion` members except `TLSVersion.TLSv1_2` and `TLSVersion.TLSv1_3` are deprecated.

19.3.2 SSL Sockets

class `ssl.SSLSocket`(*socket.socket*)

SSL sockets provide the following methods of *Socket Objects* :

— `accept()`
— `bind()`
— `close()`
— `connect()`
— `detach()`
— `fileno()`
— `getpeername()`, `getsockname()`
— `getsockopt()`, `setsockopt()`
— `gettimeout()`, `settimeout()`, `setblocking()`
— `listen()`
— `makefile()`
— `recv()`, `recv_into()` (but passing a non-zero `flags` argument is not allowed)
— `send()`, `sendall()` (with the same limitation)
— `sendfile()` (but `os.sendfile` will be used for plain-text sockets only, else `send()` will be used)
— `shutdown()`

However, since the SSL (and TLS) protocol has its own framing atop of TCP, the SSL sockets abstraction can, in certain respects, diverge from the specification of normal, OS-level sockets. See especially the *notes on non-blocking sockets*.

Instances of `SSLSocket` must be created using the `SSLContext.wrap_socket()` method.

Modifié dans la version 3.5 : The `sendfile()` method was added.

Modifié dans la version 3.5 : The `shutdown()` does not reset the socket timeout each time bytes are received or sent. The socket timeout is now the maximum total duration of the shutdown.

Obsolète depuis la version 3.6 : It is deprecated to create a `SSLSocket` instance directly, use `SSLContext.wrap_socket()` to wrap a socket.

Modifié dans la version 3.7 : `SSLSocket` instances must to created with `wrap_socket()`. In earlier versions, it was possible to create instances directly. This was never documented or officially supported.

Modifié dans la version 3.10 : Python now uses `SSL_read_ex` and `SSL_write_ex` internally. The functions support reading and writing of data larger than 2 GB. Writing zero-length data no longer fails with a protocol violation error.

SSL sockets also have the following additional methods and attributes :

`SSLSocket.read`(*len=1024*, *buffer=None*)

Read up to *len* bytes of data from the SSL socket and return the result as a `bytes` instance. If *buffer* is specified, then read into the buffer instead, and return the number of bytes read.

Raise `SSLWantReadError` or `SSLWantWriteError` if the socket is *non-blocking* and the read would block.

As at any time a re-negotiation is possible, a call to `read()` can also cause write operations.

Modifié dans la version 3.5 : The socket timeout is no longer reset each time bytes are received or sent. The socket timeout is now the maximum total duration to read up to *len* bytes.

Obsolète depuis la version 3.6 : Use `recv()` instead of `read()`.

SSLSocket.**write**(*buf*)

Write *buf* to the SSL socket and return the number of bytes written. The *buf* argument must be an object supporting the buffer interface.

Raise `SSLWantReadError` or `SSLWantWriteError` if the socket is *non-blocking* and the write would block.

As at any time a re-negotiation is possible, a call to `write()` can also cause read operations.

Modifié dans la version 3.5 : The socket timeout is no longer reset each time bytes are received or sent. The socket timeout is now the maximum total duration to write *buf*.

Obsolète depuis la version 3.6 : Use `send()` instead of `write()`.

Note

The `read()` and `write()` methods are the low-level methods that read and write unencrypted, application-level data and decrypt/encrypt it to encrypted, wire-level data. These methods require an active SSL connection, i.e. the handshake was completed and `SSLSocket.unwrap()` was not called.

Normally you should use the socket API methods like `recv()` and `send()` instead of these methods.

SSLSocket.**do_handshake**()

Perform the SSL setup handshake.

Modifié dans la version 3.4 : The handshake method also performs `match_hostname()` when the `check_hostname` attribute of the socket's `context` is true.

Modifié dans la version 3.5 : The socket timeout is no longer reset each time bytes are received or sent. The socket timeout is now the maximum total duration of the handshake.

Modifié dans la version 3.7 : Hostname or IP address is matched by OpenSSL during handshake. The function `match_hostname()` is no longer used. In case OpenSSL refuses a hostname or IP address, the handshake is aborted early and a TLS alert message is sent to the peer.

SSLSocket.**getpeercert**(*binary_form=False*)

If there is no certificate for the peer on the other end of the connection, return `None`. If the SSL handshake hasn't been done yet, raise `ValueError`.

If the `binary_form` parameter is `False`, and a certificate was received from the peer, this method returns a `dict` instance. If the certificate was not validated, the dict is empty. If the certificate was validated, it returns a dict with several keys, amongst them `subject` (the principal for which the certificate was issued) and `issuer` (the principal issuing the certificate). If a certificate contains an instance of the *Subject Alternative Name* extension (see **RFC 3280**), there will also be a `subjectAltName` key in the dictionary.

The `subject` and `issuer` fields are tuples containing the sequence of relative distinguished names (RDNs) given in the certificate's data structure for the respective fields, and each RDN is a sequence of name-value pairs. Here is a real-world example :

```
{'issuer': ((('countryName', 'IL'),),
            (('organizationName', 'StartCom Ltd.'),),
            (('organizationalUnitName',
              'Secure Digital Certificate Signing'),),
            (('commonName',
              'StartCom Class 2 Primary Intermediate Server CA'),)),
 'notAfter': 'Nov 22 08:15:19 2013 GMT',
 'notBefore': 'Nov 21 03:09:52 2011 GMT',
 'serialNumber': '95F0',
 'subject': ((('description', '571208-SLe257oHY9fVQ07Z'),),
             (('countryName', 'US'),),
             (('stateOrProvinceName', 'California'),),
             (('localityName', 'San Francisco'),),
```

(suite sur la page suivante)

(suite de la page précédente)

```
            (('organizationName', 'Electronic Frontier Foundation, Inc.'),),
            (('commonName', '*.eff.org'),),
            (('emailAddress', 'hostmaster@eff.org'),)),
 'subjectAltName': (('DNS', '*.eff.org'), ('DNS', 'eff.org')),
 'version': 3}
```

If the `binary_form` parameter is *True*, and a certificate was provided, this method returns the DER-encoded form of the entire certificate as a sequence of bytes, or *None* if the peer did not provide a certificate. Whether the peer provides a certificate depends on the SSL socket's role :

— for a client SSL socket, the server will always provide a certificate, regardless of whether validation was required ;

— for a server SSL socket, the client will only provide a certificate when requested by the server ; therefore *getpeercert()* will return *None* if you used *CERT_NONE* (rather than *CERT_OPTIONAL* or *CERT_REQUIRED*).

See also *SSLContext.check_hostname*.

Modifié dans la version 3.2 : The returned dictionary includes additional items such as `issuer` and `notBefore`.

Modifié dans la version 3.4 : *ValueError* is raised when the handshake isn't done. The returned dictionary includes additional X509v3 extension items such as `crlDistributionPoints`, `caIssuers` and `OCSP` URIs.

Modifié dans la version 3.9 : IPv6 address strings no longer have a trailing new line.

`SSLSocket.`**`get_verified_chain`**`()`

Returns verified certificate chain provided by the other end of the SSL channel as a list of DER-encoded bytes. If certificate verification was disabled method acts the same as *get_unverified_chain()*.

Ajouté dans la version 3.13.

`SSLSocket.`**`get_unverified_chain`**`()`

Returns raw certificate chain provided by the other end of the SSL channel as a list of DER-encoded bytes.

Ajouté dans la version 3.13.

`SSLSocket.`**`cipher`**`()`

Returns a three-value tuple containing the name of the cipher being used, the version of the SSL protocol that defines its use, and the number of secret bits being used. If no connection has been established, returns `None`.

`SSLSocket.`**`shared_ciphers`**`()`

Return the list of ciphers available in both the client and server. Each entry of the returned list is a three-value tuple containing the name of the cipher, the version of the SSL protocol that defines its use, and the number of secret bits the cipher uses. *shared_ciphers()* returns `None` if no connection has been established or the socket is a client socket.

Ajouté dans la version 3.5.

`SSLSocket.`**`compression`**`()`

Return the compression algorithm being used as a string, or `None` if the connection isn't compressed.

If the higher-level protocol supports its own compression mechanism, you can use *OP_NO_COMPRESSION* to disable SSL-level compression.

Ajouté dans la version 3.3.

`SSLSocket.`**`get_channel_binding`**`(`*cb_type='tls-unique'*`)`

Get channel binding data for current connection, as a bytes object. Returns `None` if not connected or the handshake has not been completed.

The *cb_type* parameter allow selection of the desired channel binding type. Valid channel binding types are listed in the *CHANNEL_BINDING_TYPES* list. Currently only the 'tls-unique' channel binding, defined by **RFC 5929**, is supported. *ValueError* will be raised if an unsupported channel binding type is requested.

Ajouté dans la version 3.3.

`SSLSocket.selected_alpn_protocol()`

Return the protocol that was selected during the TLS handshake. If `SSLContext.set_alpn_protocols()` was not called, if the other party does not support ALPN, if this socket does not support any of the client's proposed protocols, or if the handshake has not happened yet, `None` is returned.

Ajouté dans la version 3.5.

`SSLSocket.selected_npn_protocol()`

Return the higher-level protocol that was selected during the TLS/SSL handshake. If `SSLContext.set_npn_protocols()` was not called, or if the other party does not support NPN, or if the handshake has not yet happened, this will return `None`.

Ajouté dans la version 3.3.

Obsolète depuis la version 3.10 : NPN has been superseded by ALPN

`SSLSocket.unwrap()`

Performs the SSL shutdown handshake, which removes the TLS layer from the underlying socket, and returns the underlying socket object. This can be used to go from encrypted operation over a connection to unencrypted. The returned socket should always be used for further communication with the other side of the connection, rather than the original socket.

`SSLSocket.verify_client_post_handshake()`

Requests post-handshake authentication (PHA) from a TLS 1.3 client. PHA can only be initiated for a TLS 1.3 connection from a server-side socket, after the initial TLS handshake and with PHA enabled on both sides, see `SSLContext.post_handshake_auth`.

The method does not perform a cert exchange immediately. The server-side sends a CertificateRequest during the next write event and expects the client to respond with a certificate on the next read event.

If any precondition isn't met (e.g. not TLS 1.3, PHA not enabled), an `SSLError` is raised.

> **Note**
>
> Only available with OpenSSL 1.1.1 and TLS 1.3 enabled. Without TLS 1.3 support, the method raises `NotImplementedError`.

Ajouté dans la version 3.8.

`SSLSocket.version()`

Return the actual SSL protocol version negotiated by the connection as a string, or `None` if no secure connection is established. As of this writing, possible return values include `"SSLv2"`, `"SSLv3"`, `"TLSv1"`, `"TLSv1.1"` and `"TLSv1.2"`. Recent OpenSSL versions may define more return values.

Ajouté dans la version 3.5.

`SSLSocket.pending()`

Returns the number of already decrypted bytes available for read, pending on the connection.

`SSLSocket.context`

The `SSLContext` object this SSL socket is tied to.

Ajouté dans la version 3.2.

`SSLSocket.server_side`

A boolean which is `True` for server-side sockets and `False` for client-side sockets.

Ajouté dans la version 3.2.

`SSLSocket.server_hostname`

Hostname of the server : `str` type, or `None` for server-side socket or if the hostname was not specified in the constructor.

Ajouté dans la version 3.2.

Modifié dans la version 3.7 : The attribute is now always ASCII text. When `server_hostname` is an internationalized domain name (IDN), this attribute now stores the A-label form (`"xn--pythn-mua.org"`), rather than the U-label form (`"pythön.org"`).

`SSLSocket.`**`session`**

The `SSLSession` for this SSL connection. The session is available for client and server side sockets after the TLS handshake has been performed. For client sockets the session can be set before `do_handshake()` has been called to reuse a session.

Ajouté dans la version 3.6.

`SSLSocket.`**`session_reused`**

Ajouté dans la version 3.6.

19.3.3 SSL Contexts

Ajouté dans la version 3.2.

An SSL context holds various data longer-lived than single SSL connections, such as SSL configuration options, certificate(s) and private key(s). It also manages a cache of SSL sessions for server-side sockets, in order to speed up repeated connections from the same clients.

class `ssl.`**`SSLContext`**(*protocol=None*)

Create a new SSL context. You may pass *protocol* which must be one of the `PROTOCOL_*` constants defined in this module. The parameter specifies which version of the SSL protocol to use. Typically, the server chooses a particular protocol version, and the client must adapt to the server's choice. Most of the versions are not interoperable with the other versions. If not specified, the default is `PROTOCOL_TLS`; it provides the most compatibility with other versions.

Here's a table showing which versions in a client (down the side) can connect to which versions in a server (along the top) :

client / **server**	**SSLv2**	**SSLv3**	**TLS** [3]	**TLSv1**	**TLSv1.1**	**TLSv1.2**
SSLv2	oui	non	no [1]	non	non	non
SSLv3	non	oui	no [2]	non	non	non
TLS (SSLv23)[3]	no[1]	no[2]	oui	oui	oui	oui
TLSv1	non	non	oui	oui	non	non
TLSv1.1	non	non	oui	non	oui	non
TLSv1.2	non	non	oui	non	non	oui

Notes

Voir aussi

`create_default_context()` lets the `ssl` module choose security settings for a given purpose.

Modifié dans la version 3.6 : The context is created with secure default values. The options `OP_NO_COMPRESSION`, `OP_CIPHER_SERVER_PREFERENCE`, `OP_SINGLE_DH_USE`, `OP_SINGLE_ECDH_USE`, `OP_NO_SSLv2`, and `OP_NO_SSLv3` (except for `PROTOCOL_SSLv3`) are set by default. The initial cipher suite list contains only `HIGH` ciphers, no `NULL` ciphers and no `MD5` ciphers.

Obsolète depuis la version 3.10 : `SSLContext` without protocol argument is deprecated. The context class will either require `PROTOCOL_TLS_CLIENT` or `PROTOCOL_TLS_SERVER` protocol in the future.

3. TLS 1.3 protocol will be available with `PROTOCOL_TLS` in OpenSSL >= 1.1.1. There is no dedicated PROTOCOL constant for just TLS 1.3.
1. `SSLContext` disables SSLv2 with `OP_NO_SSLv2` by default.
2. `SSLContext` disables SSLv3 with `OP_NO_SSLv3` by default.

Modifié dans la version 3.10 : The default cipher suites now include only secure AES and ChaCha20 ciphers with forward secrecy and security level 2. RSA and DH keys with less than 2048 bits and ECC keys with less than 224 bits are prohibited. `PROTOCOL_TLS`, `PROTOCOL_TLS_CLIENT`, and `PROTOCOL_TLS_SERVER` use TLS 1.2 as minimum TLS version.

> **Note**
>
> `SSLContext` only supports limited mutation once it has been used by a connection. Adding new certificates to the internal trust store is allowed, but changing ciphers, verification settings, or mTLS certificates may result in surprising behavior.

> **Note**
>
> `SSLContext` is designed to be shared and used by multiple connections. Thus, it is thread-safe as long as it is not reconfigured after being used by a connection.

`SSLContext` objects have the following methods and attributes :

`SSLContext.`**`cert_store_stats`**`()`

Get statistics about quantities of loaded X.509 certificates, count of X.509 certificates flagged as CA certificates and certificate revocation lists as dictionary.

Example for a context with one CA cert and one other cert :

```
>>> context.cert_store_stats()
{'crl': 0, 'x509_ca': 1, 'x509': 2}
```

Ajouté dans la version 3.4.

`SSLContext.`**`load_cert_chain`**`(`*certfile*, *keyfile=None*, *password=None*`)`

Load a private key and the corresponding certificate. The *certfile* string must be the path to a single file in PEM format containing the certificate as well as any number of CA certificates needed to establish the certificate's authenticity. The *keyfile* string, if present, must point to a file containing the private key. Otherwise the private key will be taken from *certfile* as well. See the discussion of *Certificates* for more information on how the certificate is stored in the *certfile*.

The *password* argument may be a function to call to get the password for decrypting the private key. It will only be called if the private key is encrypted and a password is necessary. It will be called with no arguments, and it should return a string, bytes, or bytearray. If the return value is a string it will be encoded as UTF-8 before using it to decrypt the key. Alternatively a string, bytes, or bytearray value may be supplied directly as the *password* argument. It will be ignored if the private key is not encrypted and no password is needed.

If the *password* argument is not specified and a password is required, OpenSSL's built-in password prompting mechanism will be used to interactively prompt the user for a password.

An `SSLError` is raised if the private key doesn't match with the certificate.

Modifié dans la version 3.3 : New optional argument *password*.

`SSLContext.`**`load_default_certs`**`(`*purpose=Purpose.SERVER_AUTH*`)`

Load a set of default "certification authority" (CA) certificates from default locations. On Windows it loads CA certs from the `CA` and `ROOT` system stores. On all systems it calls `SSLContext.set_default_verify_paths()`. In the future the method may load CA certificates from other locations, too.

The *purpose* flag specifies what kind of CA certificates are loaded. The default settings `Purpose.SERVER_AUTH` loads certificates, that are flagged and trusted for TLS web server authentication (client side sockets). `Purpose.CLIENT_AUTH` loads CA certificates for client certificate verification on the server side.

Ajouté dans la version 3.4.

`SSLContext.load_verify_locations(cafile=None, capath=None, cadata=None)`

Load a set of "certification authority" (CA) certificates used to validate other peers' certificates when `verify_mode` is other than `CERT_NONE`. At least one of *cafile* or *capath* must be specified.

This method can also load certification revocation lists (CRLs) in PEM or DER format. In order to make use of CRLs, `SSLContext.verify_flags` must be configured properly.

The *cafile* string, if present, is the path to a file of concatenated CA certificates in PEM format. See the discussion of *Certificates* for more information about how to arrange the certificates in this file.

The *capath* string, if present, is the path to a directory containing several CA certificates in PEM format, following an OpenSSL specific layout.

The *cadata* object, if present, is either an ASCII string of one or more PEM-encoded certificates or a *bytes-like object* of DER-encoded certificates. Like with *capath* extra lines around PEM-encoded certificates are ignored but at least one certificate must be present.

Modifié dans la version 3.4 : New optional argument *cadata*

`SSLContext.get_ca_certs(binary_form=False)`

Get a list of loaded "certification authority" (CA) certificates. If the `binary_form` parameter is `False` each list entry is a dict like the output of `SSLSocket.getpeercert()`. Otherwise the method returns a list of DER-encoded certificates. The returned list does not contain certificates from *capath* unless a certificate was requested and loaded by a SSL connection.

> **Note**
>
> Certificates in a capath directory aren't loaded unless they have been used at least once.

Ajouté dans la version 3.4.

`SSLContext.get_ciphers()`

Get a list of enabled ciphers. The list is in order of cipher priority. See `SSLContext.set_ciphers()`.

Exemple :

```
>>> ctx = ssl.SSLContext(ssl.PROTOCOL_SSLv23)
>>> ctx.set_ciphers('ECDHE+AESGCM:!ECDSA')
>>> ctx.get_ciphers()
[{'aead': True,
  'alg_bits': 256,
  'auth': 'auth-rsa',
  'description': 'ECDHE-RSA-AES256-GCM-SHA384 TLSv1.2 Kx=ECDH     Au=RSA  '
                 'Enc=AESGCM(256) Mac=AEAD',
  'digest': None,
  'id': 50380848,
  'kea': 'kx-ecdhe',
  'name': 'ECDHE-RSA-AES256-GCM-SHA384',
  'protocol': 'TLSv1.2',
  'strength_bits': 256,
  'symmetric': 'aes-256-gcm'},
 {'aead': True,
  'alg_bits': 128,
  'auth': 'auth-rsa',
  'description': 'ECDHE-RSA-AES128-GCM-SHA256 TLSv1.2 Kx=ECDH     Au=RSA  '
                 'Enc=AESGCM(128) Mac=AEAD',
  'digest': None,
  'id': 50380847,
  'kea': 'kx-ecdhe',
  'name': 'ECDHE-RSA-AES128-GCM-SHA256',
```

(suite sur la page suivante)

(suite de la page précédente)

```
 'protocol': 'TLSv1.2',
 'strength_bits': 128,
 'symmetric': 'aes-128-gcm'}]
```

Ajouté dans la version 3.6.

`SSLContext.`**`set_default_verify_paths`**`()`

Load a set of default "certification authority" (CA) certificates from a filesystem path defined when building the OpenSSL library. Unfortunately, there's no easy way to know whether this method succeeds : no error is returned if no certificates are to be found. When the OpenSSL library is provided as part of the operating system, though, it is likely to be configured properly.

`SSLContext.`**`set_ciphers`**`(`*ciphers*`)`

Set the available ciphers for sockets created with this context. It should be a string in the OpenSSL cipher list format. If no cipher can be selected (because compile-time options or other configuration forbids use of all the specified ciphers), an `SSLError` will be raised.

> **Note**
>
> when connected, the `SSLSocket.cipher()` method of SSL sockets will give the currently selected cipher.
>
> TLS 1.3 cipher suites cannot be disabled with `set_ciphers()`.

`SSLContext.`**`set_alpn_protocols`**`(`*protocols*`)`

Specify which protocols the socket should advertise during the SSL/TLS handshake. It should be a list of ASCII strings, like `['http/1.1', 'spdy/2']`, ordered by preference. The selection of a protocol will happen during the handshake, and will play out according to **RFC 7301**. After a successful handshake, the `SSLSocket.selected_alpn_protocol()` method will return the agreed-upon protocol.

This method will raise `NotImplementedError` if `HAS_ALPN` is `False`.

Ajouté dans la version 3.5.

`SSLContext.`**`set_npn_protocols`**`(`*protocols*`)`

Specify which protocols the socket should advertise during the SSL/TLS handshake. It should be a list of strings, like `['http/1.1', 'spdy/2']`, ordered by preference. The selection of a protocol will happen during the handshake, and will play out according to the Application Layer Protocol Negotiation. After a successful handshake, the `SSLSocket.selected_npn_protocol()` method will return the agreed-upon protocol.

This method will raise `NotImplementedError` if `HAS_NPN` is `False`.

Ajouté dans la version 3.3.

Obsolète depuis la version 3.10 : NPN has been superseded by ALPN

`SSLContext.`**`sni_callback`**

Register a callback function that will be called after the TLS Client Hello handshake message has been received by the SSL/TLS server when the TLS client specifies a server name indication. The server name indication mechanism is specified in **RFC 6066** section 3 - Server Name Indication.

Only one callback can be set per `SSLContext`. If *sni_callback* is set to `None` then the callback is disabled. Calling this function a subsequent time will disable the previously registered callback.

The callback function will be called with three arguments ; the first being the `ssl.SSLSocket`, the second is a string that represents the server name that the client is intending to communicate (or `None` if the TLS Client Hello does not contain a server name) and the third argument is the original `SSLContext`. The server name argument is text. For internationalized domain name, the server name is an IDN A-label (`"xn--pythn-mua.org"`).

A typical use of this callback is to change the `ssl.SSLSocket`'s `SSLSocket.context` attribute to a new object of type `SSLContext` representing a certificate chain that matches the server name.

Due to the early negotiation phase of the TLS connection, only limited methods and attributes are usable like `SSLSocket.selected_alpn_protocol()` and `SSLSocket.context`. The `SSLSocket.getpeercert()`, `SSLSocket.get_verified_chain()`, `SSLSocket.get_unverified_chain()` `SSLSocket.cipher()` and `SSLSocket.compression()` methods require that the TLS connection has progressed beyond the TLS Client Hello and therefore will not return meaningful values nor can they be called safely.

The *sni_callback* function must return `None` to allow the TLS negotiation to continue. If a TLS failure is required, a constant `ALERT_DESCRIPTION_*` can be returned. Other return values will result in a TLS fatal error with `ALERT_DESCRIPTION_INTERNAL_ERROR`.

If an exception is raised from the *sni_callback* function the TLS connection will terminate with a fatal TLS alert message `ALERT_DESCRIPTION_HANDSHAKE_FAILURE`.

This method will raise `NotImplementedError` if the OpenSSL library had OPENSSL_NO_TLSEXT defined when it was built.

Ajouté dans la version 3.7.

`SSLContext.`**`set_servername_callback`**(*server_name_callback*)

This is a legacy API retained for backwards compatibility. When possible, you should use `sni_callback` instead. The given *server_name_callback* is similar to *sni_callback*, except that when the server hostname is an IDN-encoded internationalized domain name, the *server_name_callback* receives a decoded U-label (`"pythön.org"`).

If there is a decoding error on the server name, the TLS connection will terminate with an `ALERT_DESCRIPTION_INTERNAL_ERROR` fatal TLS alert message to the client.

Ajouté dans la version 3.4.

`SSLContext.`**`load_dh_params`**(*dhfile*)

Load the key generation parameters for Diffie-Hellman (DH) key exchange. Using DH key exchange improves forward secrecy at the expense of computational resources (both on the server and on the client). The *dhfile* parameter should be the path to a file containing DH parameters in PEM format.

This setting doesn't apply to client sockets. You can also use the `OP_SINGLE_DH_USE` option to further improve security.

Ajouté dans la version 3.3.

`SSLContext.`**`set_ecdh_curve`**(*curve_name*)

Set the curve name for Elliptic Curve-based Diffie-Hellman (ECDH) key exchange. ECDH is significantly faster than regular DH while arguably as secure. The *curve_name* parameter should be a string describing a well-known elliptic curve, for example `prime256v1` for a widely supported curve.

This setting doesn't apply to client sockets. You can also use the `OP_SINGLE_ECDH_USE` option to further improve security.

This method is not available if `HAS_ECDH` is `False`.

Ajouté dans la version 3.3.

Voir aussi

SSL/TLS & Perfect Forward Secrecy
Vincent Bernat.

`SSLContext.`**`wrap_socket`**(*sock*, *server_side=False*, *do_handshake_on_connect=True*, *suppress_ragged_eofs=True*, *server_hostname=None*, *session=None*)

Wrap an existing Python socket *sock* and return an instance of `SSLContext.sslsocket_class` (default

SSLSocket). The returned SSL socket is tied to the context, its settings and certificates. *sock* must be a *SOCK_STREAM* socket ; other socket types are unsupported.

The parameter `server_side` is a boolean which identifies whether server-side or client-side behavior is desired from this socket.

For client-side sockets, the context construction is lazy ; if the underlying socket isn't connected yet, the context construction will be performed after `connect()` is called on the socket. For server-side sockets, if the socket has no remote peer, it is assumed to be a listening socket, and the server-side SSL wrapping is automatically performed on client connections accepted via the `accept()` method. The method may raise *SSLError*.

On client connections, the optional parameter *server_hostname* specifies the hostname of the service which we are connecting to. This allows a single server to host multiple SSL-based services with distinct certificates, quite similarly to HTTP virtual hosts. Specifying *server_hostname* will raise a *ValueError* if *server_side* is true.

The parameter `do_handshake_on_connect` specifies whether to do the SSL handshake automatically after doing a `socket.connect()`, or whether the application program will call it explicitly, by invoking the *SSLSocket.do_handshake()* method. Calling *SSLSocket.do_handshake()* explicitly gives the program control over the blocking behavior of the socket I/O involved in the handshake.

The parameter `suppress_ragged_eofs` specifies how the `SSLSocket.recv()` method should signal unexpected EOF from the other end of the connection. If specified as *True* (the default), it returns a normal EOF (an empty bytes object) in response to unexpected EOF errors raised from the underlying socket ; if *False*, it will raise the exceptions back to the caller.

session, see *session*.

To wrap an *SSLSocket* in another *SSLSocket*, use *SSLContext.wrap_bio()*.

Modifié dans la version 3.5 : Always allow a server_hostname to be passed, even if OpenSSL does not have SNI.

Modifié dans la version 3.6 : *session* argument was added.

Modifié dans la version 3.7 : The method returns an instance of *SSLContext.sslsocket_class* instead of hard-coded *SSLSocket*.

`SSLContext.`**`sslsocket_class`**

The return type of *SSLContext.wrap_socket()*, defaults to *SSLSocket*. The attribute can be overridden on instance of class in order to return a custom subclass of *SSLSocket*.

Ajouté dans la version 3.7.

`SSLContext.`**`wrap_bio`**(*incoming*, *outgoing*, *server_side=False*, *server_hostname=None*, *session=None*)

Wrap the BIO objects *incoming* and *outgoing* and return an instance of *SSLContext.sslobject_class* (default *SSLObject*). The SSL routines will read input data from the incoming BIO and write data to the outgoing BIO.

The *server_side*, *server_hostname* and *session* parameters have the same meaning as in *SSLContext.wrap_socket()*.

Modifié dans la version 3.6 : *session* argument was added.

Modifié dans la version 3.7 : The method returns an instance of *SSLContext.sslobject_class* instead of hard-coded *SSLObject*.

`SSLContext.`**`sslobject_class`**

The return type of *SSLContext.wrap_bio()*, defaults to *SSLObject*. The attribute can be overridden on instance of class in order to return a custom subclass of *SSLObject*.

Ajouté dans la version 3.7.

`SSLContext.`**`session_stats`**()

Get statistics about the SSL sessions created or managed by this context. A dictionary is returned which maps the names of each piece of information to their numeric values. For example, here is the total number of hits and misses in the session cache since the context was created :

```
>>> stats = context.session_stats()
>>> stats['hits'], stats['misses']
(0, 0)
```

SSLContext.**check_hostname**

Whether to match the peer cert's hostname in `SSLSocket.do_handshake()`. The context's `verify_mode` must be set to `CERT_OPTIONAL` or `CERT_REQUIRED`, and you must pass *server_hostname* to `wrap_socket()` in order to match the hostname. Enabling hostname checking automatically sets `verify_mode` from `CERT_NONE` to `CERT_REQUIRED`. It cannot be set back to `CERT_NONE` as long as hostname checking is enabled. The `PROTOCOL_TLS_CLIENT` protocol enables hostname checking by default. With other protocols, hostname checking must be enabled explicitly.

Exemple :

```
import socket, ssl

context = ssl.SSLContext(ssl.PROTOCOL_TLSv1_2)
context.verify_mode = ssl.CERT_REQUIRED
context.check_hostname = True
context.load_default_certs()

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
ssl_sock = context.wrap_socket(s, server_hostname='www.verisign.com')
ssl_sock.connect(('www.verisign.com', 443))
```

Ajouté dans la version 3.4.

Modifié dans la version 3.7 : `verify_mode` is now automatically changed to `CERT_REQUIRED` when hostname checking is enabled and `verify_mode` is `CERT_NONE`. Previously the same operation would have failed with a `ValueError`.

SSLContext.**keylog_filename**

Write TLS keys to a keylog file, whenever key material is generated or received. The keylog file is designed for debugging purposes only. The file format is specified by NSS and used by many traffic analyzers such as Wireshark. The log file is opened in append-only mode. Writes are synchronized between threads, but not between processes.

Ajouté dans la version 3.8.

SSLContext.**maximum_version**

A `TLSVersion` enum member representing the highest supported TLS version. The value defaults to `TLSVersion.MAXIMUM_SUPPORTED`. The attribute is read-only for protocols other than `PROTOCOL_TLS`, `PROTOCOL_TLS_CLIENT`, and `PROTOCOL_TLS_SERVER`.

The attributes `maximum_version`, `minimum_version` and `SSLContext.options` all affect the supported SSL and TLS versions of the context. The implementation does not prevent invalid combination. For example a context with `OP_NO_TLSv1_2` in `options` and `maximum_version` set to `TLSVersion.TLSv1_2` will not be able to establish a TLS 1.2 connection.

Ajouté dans la version 3.7.

SSLContext.**minimum_version**

Like `SSLContext.maximum_version` except it is the lowest supported version or `TLSVersion.MINIMUM_SUPPORTED`.

Ajouté dans la version 3.7.

SSLContext.**num_tickets**

Control the number of TLS 1.3 session tickets of a `PROTOCOL_TLS_SERVER` context. The setting has no impact on TLS 1.0 to 1.2 connections.

Ajouté dans la version 3.8.

SSLContext.**options**

An integer representing the set of SSL options enabled on this context. The default value is *OP_ALL*, but you can specify other options such as *OP_NO_SSLv2* by ORing them together.

Modifié dans la version 3.6 : *SSLContext.options* returns *Options* flags :

```
>>> ssl.create_default_context().options
<Options.OP_ALL|OP_NO_SSLv3|OP_NO_SSLv2|OP_NO_COMPRESSION: 2197947391>
```

Obsolète depuis la version 3.7 : All `OP_NO_SSL*` and `OP_NO_TLS*` options have been deprecated since Python 3.7. Use *SSLContext.minimum_version* and *SSLContext.maximum_version* instead.

SSLContext.**post_handshake_auth**

Enable TLS 1.3 post-handshake client authentication. Post-handshake auth is disabled by default and a server can only request a TLS client certificate during the initial handshake. When enabled, a server may request a TLS client certificate at any time after the handshake.

When enabled on client-side sockets, the client signals the server that it supports post-handshake authentication.

When enabled on server-side sockets, *SSLContext.verify_mode* must be set to *CERT_OPTIONAL* or *CERT_REQUIRED*, too. The actual client cert exchange is delayed until *SSLSocket.verify_client_post_handshake()* is called and some I/O is performed.

Ajouté dans la version 3.8.

SSLContext.**protocol**

The protocol version chosen when constructing the context. This attribute is read-only.

SSLContext.**hostname_checks_common_name**

Whether *check_hostname* falls back to verify the cert's subject common name in the absence of a subject alternative name extension (default : true).

Ajouté dans la version 3.7.

Modifié dans la version 3.10 : The flag had no effect with OpenSSL before version 1.1.1l. Python 3.8.9, 3.9.3, and 3.10 include workarounds for previous versions.

SSLContext.**security_level**

An integer representing the security level for the context. This attribute is read-only.

Ajouté dans la version 3.10.

SSLContext.**verify_flags**

The flags for certificate verification operations. You can set flags like *VERIFY_CRL_CHECK_LEAF* by ORing them together. By default OpenSSL does neither require nor verify certificate revocation lists (CRLs).

Ajouté dans la version 3.4.

Modifié dans la version 3.6 : *SSLContext.verify_flags* returns *VerifyFlags* flags :

```
>>> ssl.create_default_context().verify_flags
<VerifyFlags.VERIFY_X509_TRUSTED_FIRST: 32768>
```

SSLContext.**verify_mode**

Whether to try to verify other peers' certificates and how to behave if verification fails. This attribute must be one of *CERT_NONE*, *CERT_OPTIONAL* or *CERT_REQUIRED*.

Modifié dans la version 3.6 : *SSLContext.verify_mode* returns *VerifyMode* enum :

```
>>> ssl.create_default_context().verify_mode
<VerifyMode.CERT_REQUIRED: 2>
```

SSLContext.**set_psk_client_callback**(*callback*)

Enables TLS-PSK (pre-shared key) authentication on a client-side connection.

In general, certificate based authentication should be preferred over this method.

The parameter `callback` is a callable object with the signature : `def callback(hint: str | None) -> tuple[str | None, bytes]`. The `hint` parameter is an optional identity hint sent by the server. The return value is a tuple in the form (client-identity, psk). Client-identity is an optional string which may be used by the server to select a corresponding PSK for the client. The string must be less than or equal to `256` octets when UTF-8 encoded. PSK is a *bytes-like object* representing the pre-shared key. Return a zero length PSK to reject the connection.

Setting `callback` to `None` removes any existing callback.

> **Note**
>
> When using TLS 1.3 :
> — the `hint` parameter is always `None`.
> — client-identity must be a non-empty string.

Example usage :

```
context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
context.check_hostname = False
context.verify_mode = ssl.CERT_NONE
context.maximum_version = ssl.TLSVersion.TLSv1_2
context.set_ciphers('PSK')

# A simple lambda:
psk = bytes.fromhex('c0ffee')
context.set_psk_client_callback(lambda hint: (None, psk))

# A table using the hint from the server:
psk_table = { 'ServerId_1': bytes.fromhex('c0ffee'),
              'ServerId_2': bytes.fromhex('facade')
}
def callback(hint):
    return 'ClientId_1', psk_table.get(hint, b'')
context.set_psk_client_callback(callback)
```

This method will raise `NotImplementedError` if `HAS_PSK` is `False`.

Ajouté dans la version 3.13.

`SSLContext.`**`set_psk_server_callback`**(*callback*, *identity_hint=None*)

Enables TLS-PSK (pre-shared key) authentication on a server-side connection.

In general, certificate based authentication should be preferred over this method.

The parameter `callback` is a callable object with the signature : `def callback(identity: str | None) -> bytes`. The `identity` parameter is an optional identity sent by the client which can be used to select a corresponding PSK. The return value is a *bytes-like object* representing the pre-shared key. Return a zero length PSK to reject the connection.

Setting `callback` to `None` removes any existing callback.

The parameter `identity_hint` is an optional identity hint string sent to the client. The string must be less than or equal to `256` octets when UTF-8 encoded.

> **Note**
>
> When using TLS 1.3 the `identity_hint` parameter is not sent to the client.

Example usage :

```
context = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)
context.maximum_version = ssl.TLSVersion.TLSv1_2
context.set_ciphers('PSK')

# A simple lambda:
psk = bytes.fromhex('c0ffee')
context.set_psk_server_callback(lambda identity: psk)

# A table using the identity of the client:
psk_table = { 'ClientId_1': bytes.fromhex('c0ffee'),
              'ClientId_2': bytes.fromhex('facade')
}
def callback(identity):
    return psk_table.get(identity, b'')
context.set_psk_server_callback(callback, 'ServerId_1')
```

This method will raise `NotImplementedError` if `HAS_PSK` is `False`.

Ajouté dans la version 3.13.

19.3.4 Certificates

Certificates in general are part of a public-key / private-key system. In this system, each *principal*, (which may be a machine, or a person, or an organization) is assigned a unique two-part encryption key. One part of the key is public, and is called the *public key* ; the other part is kept secret, and is called the *private key*. The two parts are related, in that if you encrypt a message with one of the parts, you can decrypt it with the other part, and **only** with the other part.

A certificate contains information about two principals. It contains the name of a *subject*, and the subject's public key. It also contains a statement by a second principal, the *issuer*, that the subject is who they claim to be, and that this is indeed the subject's public key. The issuer's statement is signed with the issuer's private key, which only the issuer knows. However, anyone can verify the issuer's statement by finding the issuer's public key, decrypting the statement with it, and comparing it to the other information in the certificate. The certificate also contains information about the time period over which it is valid. This is expressed as two fields, called "notBefore" and "notAfter".

In the Python use of certificates, a client or server can use a certificate to prove who they are. The other side of a network connection can also be required to produce a certificate, and that certificate can be validated to the satisfaction of the client or server that requires such validation. The connection attempt can be set to raise an exception if the validation fails. Validation is done automatically, by the underlying OpenSSL framework ; the application need not concern itself with its mechanics. But the application does usually need to provide sets of certificates to allow this process to take place.

Python uses files to contain certificates. They should be formatted as "PEM" (see **RFC 1422**), which is a base-64 encoded form wrapped with a header line and a footer line :

```
-----BEGIN CERTIFICATE-----
... (certificate in base64 PEM encoding) ...
-----END CERTIFICATE-----
```

Certificate chains

The Python files which contain certificates can contain a sequence of certificates, sometimes called a *certificate chain*. This chain should start with the specific certificate for the principal who "is" the client or server, and then the certificate for the issuer of that certificate, and then the certificate for the issuer of *that* certificate, and so on up the chain till you get to a certificate which is *self-signed*, that is, a certificate which has the same subject and issuer, sometimes called a *root certificate*. The certificates should just be concatenated together in the certificate file. For example, suppose we had a three certificate chain, from our server certificate to the certificate of the certification authority that signed our server certificate, to the root certificate of the agency which issued the certification authority's certificate :

```
-----BEGIN CERTIFICATE-----
... (certificate for your server)...
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
... (the certificate for the CA)...
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
... (the root certificate for the CA's issuer)...
-----END CERTIFICATE-----
```

CA certificates

If you are going to require validation of the other side of the connection's certificate, you need to provide a "CA certs" file, filled with the certificate chains for each issuer you are willing to trust. Again, this file just contains these chains concatenated together. For validation, Python will use the first chain it finds in the file which matches. The platform's certificates file can be used by calling `SSLContext.load_default_certs()`, this is done automatically with `create_default_context()`.

Combined key and certificate

Often the private key is stored in the same file as the certificate; in this case, only the `certfile` parameter to `SSLContext.load_cert_chain()` needs to be passed. If the private key is stored with the certificate, it should come before the first certificate in the certificate chain :

```
-----BEGIN RSA PRIVATE KEY-----
... (private key in base64 encoding) ...
-----END RSA PRIVATE KEY-----
-----BEGIN CERTIFICATE-----
... (certificate in base64 PEM encoding) ...
-----END CERTIFICATE-----
```

Self-signed certificates

If you are going to create a server that provides SSL-encrypted connection services, you will need to acquire a certificate for that service. There are many ways of acquiring appropriate certificates, such as buying one from a certification authority. Another common practice is to generate a self-signed certificate. The simplest way to do this is with the OpenSSL package, using something like the following :

```
% openssl req -new -x509 -days 365 -nodes -out cert.pem -keyout cert.pem
Generating a 1024 bit RSA private key
.......++++++
.............................++++++
writing new private key to 'cert.pem'
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:MyState
Locality Name (eg, city) []:Some City
Organization Name (eg, company) [Internet Widgits Pty Ltd]:My Organization, Inc.
Organizational Unit Name (eg, section) []:My Group
Common Name (eg, YOUR name) []:myserver.mygroup.myorganization.com
```

(suite sur la page suivante)

(suite de la page précédente)

```
Email Address []:ops@myserver.mygroup.myorganization.com
%
```

The disadvantage of a self-signed certificate is that it is its own root certificate, and no one else will have it in their cache of known (and trusted) root certificates.

19.3.5 Exemples

Testing for SSL support

To test for the presence of SSL support in a Python installation, user code should use the following idiom :

```
try:
    import ssl
except ImportError:
    pass
else:
    ...  # do something that requires SSL support
```

Client-side operation

This example creates a SSL context with the recommended security settings for client sockets, including automatic certificate verification :

```
>>> context = ssl.create_default_context()
```

If you prefer to tune security settings yourself, you might create a context from scratch (but beware that you might not get the settings right) :

```
>>> context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
>>> context.load_verify_locations("/etc/ssl/certs/ca-bundle.crt")
```

(this snippet assumes your operating system places a bundle of all CA certificates in `/etc/ssl/certs/ca-bundle.crt` ; if not, you'll get an error and have to adjust the location)

The *PROTOCOL_TLS_CLIENT* protocol configures the context for cert validation and hostname verification. *verify_mode* is set to *CERT_REQUIRED* and *check_hostname* is set to `True`. All other protocols create SSL contexts with insecure defaults.

When you use the context to connect to a server, *CERT_REQUIRED* and *check_hostname* validate the server certificate : it ensures that the server certificate was signed with one of the CA certificates, checks the signature for correctness, and verifies other properties like validity and identity of the hostname :

```
>>> conn = context.wrap_socket(socket.socket(socket.AF_INET),
...                            server_hostname="www.python.org")
>>> conn.connect(("www.python.org", 443))
```

You may then fetch the certificate :

```
>>> cert = conn.getpeercert()
```

Visual inspection shows that the certificate does identify the desired service (that is, the HTTPS host `www.python.org`) :

```
>>> pprint.pprint(cert)
{'OCSP': ('http://ocsp.digicert.com',),
 'caIssuers': ('http://cacerts.digicert.com/DigiCertSHA2ExtendedValidationServerCA.
↪crt',),
```

(suite sur la page suivante)

(suite de la page précédente)

```
 'crlDistributionPoints': ('http://crl3.digicert.com/sha2-ev-server-g1.crl',
                           'http://crl4.digicert.com/sha2-ev-server-g1.crl'),
 'issuer': ((('countryName', 'US'),),
            (('organizationName', 'DigiCert Inc'),),
            (('organizationalUnitName', 'www.digicert.com'),),
            (('commonName', 'DigiCert SHA2 Extended Validation Server CA'),)),
 'notAfter': 'Sep  9 12:00:00 2016 GMT',
 'notBefore': 'Sep  5 00:00:00 2014 GMT',
 'serialNumber': '01BB6F00122B177F36CAB49CEA8B6B26',
 'subject': ((('businessCategory', 'Private Organization'),),
             (('1.3.6.1.4.1.311.60.2.1.3', 'US'),),
             (('1.3.6.1.4.1.311.60.2.1.2', 'Delaware'),),
             (('serialNumber', '3359300'),),
             (('streetAddress', '16 Allen Rd'),),
             (('postalCode', '03894-4801'),),
             (('countryName', 'US'),),
             (('stateOrProvinceName', 'NH'),),
             (('localityName', 'Wolfeboro'),),
             (('organizationName', 'Python Software Foundation'),),
             (('commonName', 'www.python.org'),)),
 'subjectAltName': (('DNS', 'www.python.org'),
                    ('DNS', 'python.org'),
                    ('DNS', 'pypi.org'),
                    ('DNS', 'docs.python.org'),
                    ('DNS', 'testpypi.org'),
                    ('DNS', 'bugs.python.org'),
                    ('DNS', 'wiki.python.org'),
                    ('DNS', 'hg.python.org'),
                    ('DNS', 'mail.python.org'),
                    ('DNS', 'packaging.python.org'),
                    ('DNS', 'pythonhosted.org'),
                    ('DNS', 'www.pythonhosted.org'),
                    ('DNS', 'test.pythonhosted.org'),
                    ('DNS', 'us.pycon.org'),
                    ('DNS', 'id.python.org')),
 'version': 3}
```

Now the SSL channel is established and the certificate verified, you can proceed to talk with the server :

```
>>> conn.sendall(b"HEAD / HTTP/1.0\r\nHost: linuxfr.org\r\n\r\n")
>>> pprint.pprint(conn.recv(1024).split(b"\r\n"))
[b'HTTP/1.1 200 OK',
 b'Date: Sat, 18 Oct 2014 18:27:20 GMT',
 b'Server: nginx',
 b'Content-Type: text/html; charset=utf-8',
 b'X-Frame-Options: SAMEORIGIN',
 b'Content-Length: 45679',
 b'Accept-Ranges: bytes',
 b'Via: 1.1 varnish',
 b'Age: 2188',
 b'X-Served-By: cache-lcy1134-LCY',
 b'X-Cache: HIT',
 b'X-Cache-Hits: 11',
 b'Vary: Cookie',
 b'Strict-Transport-Security: max-age=63072000; includeSubDomains',
 b'Connection: close',
```

(suite sur la page suivante)

(suite de la page précédente)

```
 b'',
 b'']
```

See the discussion of *Security considerations* below.

Server-side operation

For server operation, typically you'll need to have a server certificate, and private key, each in a file. You'll first create a context holding the key and the certificate, so that clients can check your authenticity. Then you'll open a socket, bind it to a port, call `listen()` on it, and start waiting for clients to connect :

```
import socket, ssl

context = ssl.create_default_context(ssl.Purpose.CLIENT_AUTH)
context.load_cert_chain(certfile="mycertfile", keyfile="mykeyfile")

bindsocket = socket.socket()
bindsocket.bind(('myaddr.example.com', 10023))
bindsocket.listen(5)
```

When a client connects, you'll call `accept()` on the socket to get the new socket from the other end, and use the context's *`SSLContext.wrap_socket()`* method to create a server-side SSL socket for the connection :

```
while True:
    newsocket, fromaddr = bindsocket.accept()
    connstream = context.wrap_socket(newsocket, server_side=True)
    try:
        deal_with_client(connstream)
    finally:
        connstream.shutdown(socket.SHUT_RDWR)
        connstream.close()
```

Then you'll read data from the `connstream` and do something with it till you are finished with the client (or the client is finished with you) :

```
def deal_with_client(connstream):
    data = connstream.recv(1024)
    # empty data means the client is finished with us
    while data:
        if not do_something(connstream, data):
            # we'll assume do_something returns False
            # when we're finished with client
            break
        data = connstream.recv(1024)
    # finished with client
```

And go back to listening for new client connections (of course, a real server would probably handle each client connection in a separate thread, or put the sockets in *non-blocking mode* and use an event loop).

19.3.6 Notes on non-blocking sockets

SSL sockets behave slightly different than regular sockets in non-blocking mode. When working with non-blocking sockets, there are thus several things you need to be aware of :

— Most *`SSLSocket`* methods will raise either *`SSLWantWriteError`* or *`SSLWantReadError`* instead of *`BlockingIOError`* if an I/O operation would block. *`SSLWantReadError`* will be raised if a read operation on the underlying socket is necessary, and *`SSLWantWriteError`* for a write operation on the underlying socket. Note that attempts to *write* to an SSL socket may require *reading* from the underlying socket first, and attempts to *read* from the SSL socket may require a prior *write* to the underlying socket.

Modifié dans la version 3.5 : In earlier Python versions, the `SSLSocket.send()` method returned zero instead of raising `SSLWantWriteError` or `SSLWantReadError`.

— Calling `select()` tells you that the OS-level socket can be read from (or written to), but it does not imply that there is sufficient data at the upper SSL layer. For example, only part of an SSL frame might have arrived. Therefore, you must be ready to handle `SSLSocket.recv()` and `SSLSocket.send()` failures, and retry after another call to `select()`.

— Conversely, since the SSL layer has its own framing, a SSL socket may still have data available for reading without `select()` being aware of it. Therefore, you should first call `SSLSocket.recv()` to drain any potentially available data, and then only block on a `select()` call if still necessary.

(of course, similar provisions apply when using other primitives such as `poll()`, or those in the `selectors` module)

— The SSL handshake itself will be non-blocking : the `SSLSocket.do_handshake()` method has to be retried until it returns successfully. Here is a synopsis using `select()` to wait for the socket's readiness :

```
while True:
    try:
        sock.do_handshake()
        break
    except ssl.SSLWantReadError:
        select.select([sock], [], [])
    except ssl.SSLWantWriteError:
        select.select([], [sock], [])
```

Voir aussi

The `asyncio` module supports *non-blocking SSL sockets* and provides a higher level *Streams API*. It polls for events using the `selectors` module and handles `SSLWantWriteError`, `SSLWantReadError` and `BlockingIOError` exceptions. It runs the SSL handshake asynchronously as well.

19.3.7 Memory BIO Support

Ajouté dans la version 3.5.

Ever since the SSL module was introduced in Python 2.6, the `SSLSocket` class has provided two related but distinct areas of functionality :

— SSL protocol handling
— Network IO

The network IO API is identical to that provided by `socket.socket`, from which `SSLSocket` also inherits. This allows an SSL socket to be used as a drop-in replacement for a regular socket, making it very easy to add SSL support to an existing application.

Combining SSL protocol handling and network IO usually works well, but there are some cases where it doesn't. An example is async IO frameworks that want to use a different IO multiplexing model than the "select/poll on a file descriptor" (readiness based) model that is assumed by `socket.socket` and by the internal OpenSSL socket IO routines. This is mostly relevant for platforms like Windows where this model is not efficient. For this purpose, a reduced scope variant of `SSLSocket` called `SSLObject` is provided.

class ssl.**SSLObject**

A reduced-scope variant of `SSLSocket` representing an SSL protocol instance that does not contain any network IO methods. This class is typically used by framework authors that want to implement asynchronous IO for SSL through memory buffers.

This class implements an interface on top of a low-level SSL object as implemented by OpenSSL. This object captures the state of an SSL connection but does not provide any network IO itself. IO needs to be performed through separate "BIO" objects which are OpenSSL's IO abstraction layer.

This class has no public constructor. An `SSLObject` instance must be created using the `wrap_bio()` method. This method will create the `SSLObject` instance and bind it to a pair of BIOs. The *incoming* BIO is used to

pass data from Python to the SSL protocol instance, while the *outgoing* BIO is used to pass data the other way around.

The following methods are available :
— `context`
— `server_side`
— `server_hostname`
— `session`
— `session_reused`
— `read()`
— `write()`
— `getpeercert()`
— `get_verified_chain()`
— `get_unverified_chain()`
— `selected_alpn_protocol()`
— `selected_npn_protocol()`
— `cipher()`
— `shared_ciphers()`
— `compression()`
— `pending()`
— `do_handshake()`
— `verify_client_post_handshake()`
— `unwrap()`
— `get_channel_binding()`
— `version()`

When compared to `SSLSocket`, this object lacks the following features :
— Any form of network IO ; `recv()` and `send()` read and write only to the underlying `MemoryBIO` buffers.
— There is no *do_handshake_on_connect* machinery. You must always manually call `do_handshake()` to start the handshake.
— There is no handling of *suppress_ragged_eofs*. All end-of-file conditions that are in violation of the protocol are reported via the `SSLEOFError` exception.
— The method `unwrap()` call does not return anything, unlike for an SSL socket where it returns the underlying socket.
— The *server_name_callback* callback passed to `SSLContext.set_servername_callback()` will get an `SSLObject` instance instead of a `SSLSocket` instance as its first parameter.

Some notes related to the use of `SSLObject` :
— All IO on an `SSLObject` is *non-blocking*. This means that for example `read()` will raise an `SSLWantReadError` if it needs more data than the incoming BIO has available.

Modifié dans la version 3.7 : `SSLObject` instances must be created with `wrap_bio()`. In earlier versions, it was possible to create instances directly. This was never documented or officially supported.

An SSLObject communicates with the outside world using memory buffers. The class `MemoryBIO` provides a memory buffer that can be used for this purpose. It wraps an OpenSSL memory BIO (Basic IO) object :

class `ssl`.**MemoryBIO**

A memory buffer that can be used to pass data between Python and an SSL protocol instance.

pending

Return the number of bytes currently in the memory buffer.

eof

A boolean indicating whether the memory BIO is current at the end-of-file position.

read(*n=-1*)

Read up to *n* bytes from the memory buffer. If *n* is not specified or negative, all bytes are returned.

write(*buf*)

Write the bytes from *buf* to the memory BIO. The *buf* argument must be an object supporting the buffer protocol.

The return value is the number of bytes written, which is always equal to the length of *buf*.

write_eof()

Write an EOF marker to the memory BIO. After this method has been called, it is illegal to call *write()*. The attribute *eof* will become true after all data currently in the buffer has been read.

19.3.8 SSL session

Ajouté dans la version 3.6.

class ssl.**SSLSession**

Session object used by *session*.

id

time

timeout

ticket_lifetime_hint

has_ticket

19.3.9 Security considerations

Best defaults

For **client use**, if you don't have any special requirements for your security policy, it is highly recommended that you use the *create_default_context()* function to create your SSL context. It will load the system's trusted CA certificates, enable certificate validation and hostname checking, and try to choose reasonably secure protocol and cipher settings.

For example, here is how you would use the *smtplib.SMTP* class to create a trusted, secure connection to a SMTP server :

```
>>> import ssl, smtplib
>>> smtp = smtplib.SMTP("mail.python.org", port=587)
>>> context = ssl.create_default_context()
>>> smtp.starttls(context=context)
(220, b'2.0.0 Ready to start TLS')
```

If a client certificate is needed for the connection, it can be added with *SSLContext.load_cert_chain()*.

By contrast, if you create the SSL context by calling the *SSLContext* constructor yourself, it will not have certificate validation nor hostname checking enabled by default. If you do so, please read the paragraphs below to achieve a good security level.

Manual settings

Verifying certificates

When calling the *SSLContext* constructor directly, *CERT_NONE* is the default. Since it does not authenticate the other peer, it can be insecure, especially in client mode where most of the time you would like to ensure the authenticity of the server you're talking to. Therefore, when in client mode, it is highly recommended to use *CERT_REQUIRED*. However, it is in itself not sufficient ; you also have to check that the server certificate, which can be obtained by calling *SSLSocket.getpeercert()*, matches the desired service. For many protocols and applications, the service can be identified by the hostname. This common check is automatically performed when *SSLContext.check_hostname* is enabled.

Modifié dans la version 3.7 : Hostname matchings is now performed by OpenSSL. Python no longer uses `match_hostname()`.

In server mode, if you want to authenticate your clients using the SSL layer (rather than using a higher-level authentication mechanism), you'll also have to specify *CERT_REQUIRED* and similarly check the client certificate.

Protocol versions

SSL versions 2 and 3 are considered insecure and are therefore dangerous to use. If you want maximum compatibility between clients and servers, it is recommended to use *PROTOCOL_TLS_CLIENT* or *PROTOCOL_TLS_SERVER* as the protocol version. SSLv2 and SSLv3 are disabled by default.

```
>>> client_context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
>>> client_context.minimum_version = ssl.TLSVersion.TLSv1_3
>>> client_context.maximum_version = ssl.TLSVersion.TLSv1_3
```

The SSL context created above will only allow TLSv1.3 and later (if supported by your system) connections to a server. *PROTOCOL_TLS_CLIENT* implies certificate validation and hostname checks by default. You have to load certificates into the context.

Cipher selection

If you have advanced security requirements, fine-tuning of the ciphers enabled when negotiating a SSL session is possible through the *SSLContext.set_ciphers()* method. Starting from Python 3.2.3, the ssl module disables certain weak ciphers by default, but you may want to further restrict the cipher choice. Be sure to read OpenSSL's documentation about the cipher list format. If you want to check which ciphers are enabled by a given cipher list, use *SSLContext.get_ciphers()* or the `openssl ciphers` command on your system.

Multi-processing

If using this module as part of a multi-processed application (using, for example the *multiprocessing* or *concurrent.futures* modules), be aware that OpenSSL's internal random number generator does not properly handle forked processes. Applications must change the PRNG state of the parent process if they use any SSL feature with *os.fork()*. Any successful call of *RAND_add()* or *RAND_bytes()* is sufficient.

19.3.10 TLS 1.3

Ajouté dans la version 3.7.

The TLS 1.3 protocol behaves slightly differently than previous version of TLS/SSL. Some new TLS 1.3 features are not yet available.

- TLS 1.3 uses a disjunct set of cipher suites. All AES-GCM and ChaCha20 cipher suites are enabled by default. The method *SSLContext.set_ciphers()* cannot enable or disable any TLS 1.3 ciphers yet, but *SSLContext.get_ciphers()* returns them.
- Session tickets are no longer sent as part of the initial handshake and are handled differently. *SSLSocket.session* and *SSLSession* are not compatible with TLS 1.3.
- Client-side certificates are also no longer verified during the initial handshake. A server can request a certificate at any time. Clients process certificate requests while they send or receive application data from the server.
- TLS 1.3 features like early data, deferred TLS client cert request, signature algorithm configuration, and rekeying are not supported yet.

Voir aussi

Class *socket.socket*
Documentation of underlying *socket* class

SSL/TLS Strong Encryption : An Introduction
Intro from the Apache HTTP Server documentation

RFC 1422 : Privacy Enhancement for Internet Electronic Mail : Part II : Certificate-Based Key Management
Steve Kent

RFC 4086 : Randomness Requirements for Security
Donald E., Jeffrey I. Schiller

RFC 5280 : Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile
D. Cooper

RFC 5246 : The Transport Layer Security (TLS) Protocol Version 1.2
T. Dierks et. al.

RFC 6066 : Transport Layer Security (TLS) Extensions
D. Eastlake

IANA TLS : Transport Layer Security (TLS) Parameters
IANA

RFC 7525 : Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)
IETF

Mozilla's Server Side TLS recommendations
Mozilla

19.4 `select` --- Waiting for I/O completion

This module provides access to the `select()` and `poll()` functions available in most operating systems, `devpoll()` available on Solaris and derivatives, `epoll()` available on Linux 2.5+ and `kqueue()` available on most BSD. Note that on Windows, it only works for sockets ; on other operating systems, it also works for other file types (in particular, on Unix, it works on pipes). It cannot be used on regular files to determine whether a file has grown since it was last read.

> **Note**
>
> The *`selectors`* module allows high-level and efficient I/O multiplexing, built upon the *`select`* module primitives. Users are encouraged to use the *`selectors`* module instead, unless they want precise control over the OS-level primitives used.

Availability : not WASI.

This module does not work or is not available on WebAssembly. See *Plateformes WebAssembly* for more information.

Le module définit :

exception `select.`**`error`**

A deprecated alias of *`OSError`*.

Modifié dans la version 3.3 : Following **PEP 3151**, this class was made an alias of *`OSError`*.

`select.`**`devpoll`**`()`

(Only supported on Solaris and derivatives.) Returns a `/dev/poll` polling object ; see section */dev/poll Polling Objects* below for the methods supported by devpoll objects.

`devpoll()` objects are linked to the number of file descriptors allowed at the time of instantiation. If your program reduces this value, `devpoll()` will fail. If your program increases this value, `devpoll()` may return an incomplete list of active file descriptors.

The new file descriptor is *non-inheritable*.

Ajouté dans la version 3.3.

Modifié dans la version 3.4 : Le nouveau descripteur de fichier est maintenant non-héritable.

`select.`**`epoll`**`(`*sizehint=-1*, *flags=0*`)`

(Only supported on Linux 2.5.44 and newer.) Return an edge polling object, which can be used as Edge or Level Triggered interface for I/O events.

sizehint informs epoll about the expected number of events to be registered. It must be positive, or `-1` to use the default. It is only used on older systems where `epoll_create1()` is not available ; otherwise it has no effect (though its value is still checked).

flags is deprecated and completely ignored. However, when supplied, its value must be `0` or `select.EPOLL_CLOEXEC`, otherwise `OSError` is raised.

See the *Edge and Level Trigger Polling (epoll) Objects* section below for the methods supported by epolling objects.

`epoll` objects support the context management protocol : when used in a `with` statement, the new file descriptor is automatically closed at the end of the block.

The new file descriptor is *non-inheritable*.

Modifié dans la version 3.3 : Added the *flags* parameter.

Modifié dans la version 3.4 : Support for the `with` statement was added. The new file descriptor is now non-inheritable.

Obsolète depuis la version 3.4 : The *flags* parameter. `select.EPOLL_CLOEXEC` is used by default now. Use `os.set_inheritable()` to make the file descriptor inheritable.

`select.`**`poll`**`()`

(Not supported by all operating systems.) Returns a polling object, which supports registering and unregistering file descriptors, and then polling them for I/O events ; see section *Polling Objects* below for the methods supported by polling objects.

`select.`**`kqueue`**`()`

(Only supported on BSD.) Returns a kernel queue object ; see section *Kqueue Objects* below for the methods supported by kqueue objects.

The new file descriptor is *non-inheritable*.

Modifié dans la version 3.4 : Le nouveau descripteur de fichier est maintenant non-héritable.

`select.`**`kevent`**`(`*ident*, *filter=KQ_FILTER_READ*, *flags=KQ_EV_ADD*, *fflags=0*, *data=0*, *udata=0*`)`

(Only supported on BSD.) Returns a kernel event object ; see section *Kevent Objects* below for the methods supported by kevent objects.

`select.`**`select`**`(`*rlist*, *wlist*, *xlist*[, *timeout*]`)`

This is a straightforward interface to the Unix `select()` system call. The first three arguments are iterables of 'waitable objects' : either integers representing file descriptors or objects with a parameterless method named `fileno()` returning such an integer :

— *rlist* : wait until ready for reading
— *wlist* : wait until ready for writing
— *xlist* : wait for an "exceptional condition" (see the manual page for what your system considers such a condition)

Empty iterables are allowed, but acceptance of three empty iterables is platform-dependent. (It is known to work on Unix but not on Windows.) The optional *timeout* argument specifies a time-out as a floating-point number in seconds. When the *timeout* argument is omitted the function blocks until at least one file descriptor is ready. A time-out value of zero specifies a poll and never blocks.

The return value is a triple of lists of objects that are ready : subsets of the first three arguments. When the time-out is reached without a file descriptor becoming ready, three empty lists are returned.

Among the acceptable object types in the iterables are Python *file objects* (e.g. `sys.stdin`, or objects returned by `open()` or `os.popen()`), socket objects returned by `socket.socket()`. You may also define a *wrapper* class yourself, as long as it has an appropriate `fileno()` method (that really returns a file descriptor, not just a random integer).

> **Note**
>
> File objects on Windows are not acceptable, but sockets are. On Windows, the underlying `select()` function is provided by the WinSock library, and does not handle file descriptors that don't originate from WinSock.

Modifié dans la version 3.5 : The function is now retried with a recomputed timeout when interrupted by

a signal, except if the signal handler raises an exception (see **PEP 475** for the rationale), instead of raising `InterruptedError`.

`select.PIPE_BUF`

The minimum number of bytes which can be written without blocking to a pipe when the pipe has been reported as ready for writing by `select()`, `poll()` or another interface in this module. This doesn't apply to other kind of file-like objects such as sockets.

This value is guaranteed by POSIX to be at least 512.

Availability : Unix

Ajouté dans la version 3.2.

19.4.1 `/dev/poll` Polling Objects

Solaris and derivatives have `/dev/poll`. While `select()` is *O*(*highest file descriptor*) and `poll()` is *O*(*number of file descriptors*), `/dev/poll` is *O*(*active file descriptors*).

`/dev/poll` behaviour is very close to the standard `poll()` object.

`devpoll.close()`

Close the file descriptor of the polling object.

Ajouté dans la version 3.4.

`devpoll.closed`

`True` if the polling object is closed.

Ajouté dans la version 3.4.

`devpoll.fileno()`

Return the file descriptor number of the polling object.

Ajouté dans la version 3.4.

`devpoll.register(fd[, eventmask])`

Register a file descriptor with the polling object. Future calls to the `poll()` method will then check whether the file descriptor has any pending I/O events. *fd* can be either an integer, or an object with a `fileno()` method that returns an integer. File objects implement `fileno()`, so they can also be used as the argument.

eventmask is an optional bitmask describing the type of events you want to check for. The constants are the same that with `poll()` object. The default value is a combination of the constants `POLLIN`, `POLLPRI`, and `POLLOUT`.

> **Avertissement**
>
> Registering a file descriptor that's already registered is not an error, but the result is undefined. The appropriate action is to unregister or modify it first. This is an important difference compared with `poll()`.

`devpoll.modify(fd[, eventmask])`

This method does an `unregister()` followed by a `register()`. It is (a bit) more efficient that doing the same explicitly.

`devpoll.unregister(fd)`

Remove a file descriptor being tracked by a polling object. Just like the `register()` method, *fd* can be an integer or an object with a `fileno()` method that returns an integer.

Attempting to remove a file descriptor that was never registered is safely ignored.

`devpoll.poll([timeout])`

Polls the set of registered file descriptors, and returns a possibly empty list containing `(fd, event)` 2-tuples for the descriptors that have events or errors to report. *fd* is the file descriptor, and *event* is a bitmask with

bits set for the reported events for that descriptor --- `POLLIN` for waiting input, `POLLOUT` to indicate that the descriptor can be written to, and so forth. An empty list indicates that the call timed out and no file descriptors had any events to report. If *timeout* is given, it specifies the length of time in milliseconds which the system will wait for events before returning. If *timeout* is omitted, -1, or `None`, the call will block until there is an event for this poll object.

Modifié dans la version 3.5 : The function is now retried with a recomputed timeout when interrupted by a signal, except if the signal handler raises an exception (see **PEP 475** for the rationale), instead of raising `InterruptedError`.

19.4.2 Edge and Level Trigger Polling (epoll) Objects

https://linux.die.net/man/4/epoll

eventmask

Constante	Signification
`EPOLLIN`	Available for read
`EPOLLOUT`	Available for write
`EPOLLPRI`	Urgent data for read
`EPOLLERR`	Error condition happened on the assoc. fd
`EPOLLHUP`	Hang up happened on the assoc. fd
`EPOLLET`	Set Edge Trigger behavior, the default is Level Trigger behavior
`EPOLLONESH`	Set one-shot behavior. After one event is pulled out, the fd is internally disabled
`EPOLLEXCLU`	Wake only one epoll object when the associated fd has an event. The default (if this flag is not set) is to wake all epoll objects polling on a fd.
`EPOLLRDHUP`	Stream socket peer closed connection or shut down writing half of connection.
`EPOLLRDNORM`	Equivalent to `EPOLLIN`
`EPOLLRDBAND`	Priority data band can be read.
`EPOLLWRNORM`	Equivalent to `EPOLLOUT`
`EPOLLWRBAND`	Priority data may be written.
`EPOLLMSG`	Ignored.
`EPOLLWAKEUP`	Prevents sleep during event waiting.

Ajouté dans la version 3.6 : `EPOLLEXCLUSIVE` was added. It's only supported by Linux Kernel 4.5 or later.

Ajouté dans la version 3.14 : `EPOLLWAKEUP` was added. It's only supported by Linux Kernel 3.5 or later.

`epoll.`**`close`**`()`

Close the control file descriptor of the epoll object.

`epoll.`**`closed`**

`True` if the epoll object is closed.

`epoll.`**`fileno`**`()`

Return the file descriptor number of the control fd.

`epoll.`**`fromfd`**`(`*fd*`)`

Create an epoll object from a given file descriptor.

`epoll.`**`register`**`(`*fd*[, *eventmask*]`)`

Register a fd descriptor with the epoll object.

`epoll.`**`modify`**`(`*fd*, *eventmask*`)`

Modify a registered file descriptor.

`epoll.`**`unregister`**`(`*fd*`)`

Remove a registered file descriptor from the epoll object.

Modifié dans la version 3.9 : The method no longer ignores the `EBADF` error.

`epoll.poll(timeout=None, maxevents=-1)`

Wait for events. timeout in seconds (float)

Modifié dans la version 3.5 : The function is now retried with a recomputed timeout when interrupted by a signal, except if the signal handler raises an exception (see **PEP 475** for the rationale), instead of raising `InterruptedError`.

19.4.3 Polling Objects

The `poll()` system call, supported on most Unix systems, provides better scalability for network servers that service many, many clients at the same time. `poll()` scales better because the system call only requires listing the file descriptors of interest, while `select()` builds a bitmap, turns on bits for the fds of interest, and then afterward the whole bitmap has to be linearly scanned again. `select()` is *O*(*highest file descriptor*), while `poll()` is *O*(*number of file descriptors*).

`poll.register(fd[, eventmask])`

Register a file descriptor with the polling object. Future calls to the `poll()` method will then check whether the file descriptor has any pending I/O events. *fd* can be either an integer, or an object with a `fileno()` method that returns an integer. File objects implement `fileno()`, so they can also be used as the argument.

eventmask is an optional bitmask describing the type of events you want to check for, and can be a combination of the constants `POLLIN`, `POLLPRI`, and `POLLOUT`, described in the table below. If not specified, the default value used will check for all 3 types of events.

Constante	Signification
`POLLIN`	There is data to read
`POLLPRI`	There is urgent data to read
`POLLOUT`	Ready for output : writing will not block
`POLLERR`	Error condition of some sort
`POLLHUP`	Hung up
`POLLRDHUP`	Stream socket peer closed connection, or shut down writing half of connection
`POLLNVAL`	Invalid request : descriptor not open

Registering a file descriptor that's already registered is not an error, and has the same effect as registering the descriptor exactly once.

`poll.modify(fd, eventmask)`

Modifies an already registered fd. This has the same effect as `register(fd, eventmask)`. Attempting to modify a file descriptor that was never registered causes an `OSError` exception with errno `ENOENT` to be raised.

`poll.unregister(fd)`

Remove a file descriptor being tracked by a polling object. Just like the `register()` method, *fd* can be an integer or an object with a `fileno()` method that returns an integer.

Attempting to remove a file descriptor that was never registered causes a `KeyError` exception to be raised.

`poll.poll([timeout])`

Polls the set of registered file descriptors, and returns a possibly empty list containing `(fd, event)` 2-tuples for the descriptors that have events or errors to report. *fd* is the file descriptor, and *event* is a bitmask with bits set for the reported events for that descriptor --- `POLLIN` for waiting input, `POLLOUT` to indicate that the descriptor can be written to, and so forth. An empty list indicates that the call timed out and no file descriptors had any events to report. If *timeout* is given, it specifies the length of time in milliseconds which the system will wait for events before returning. If *timeout* is omitted, negative, or `None`, the call will block until there is an event for this poll object.

Modifié dans la version 3.5 : The function is now retried with a recomputed timeout when interrupted by a signal, except if the signal handler raises an exception (see **PEP 475** for the rationale), instead of raising `InterruptedError`.

19.4.4 Kqueue Objects

`kqueue.close()`

Close the control file descriptor of the kqueue object.

`kqueue.closed`

`True` if the kqueue object is closed.

`kqueue.fileno()`

Return the file descriptor number of the control fd.

`kqueue.fromfd(fd)`

Create a kqueue object from a given file descriptor.

`kqueue.control(changelist, max_events[, timeout]) → eventlist`

Low level interface to kevent

— changelist must be an iterable of kevent objects or `None`

— max_events must be 0 or a positive integer

— timeout in seconds (floats possible) ; the default is `None`, to wait forever

Modifié dans la version 3.5 : The function is now retried with a recomputed timeout when interrupted by a signal, except if the signal handler raises an exception (see **PEP 475** for the rationale), instead of raising `InterruptedError`.

19.4.5 Kevent Objects

https://man.freebsd.org/cgi/man.cgi?query=kqueue&sektion=2

`kevent.ident`

Value used to identify the event. The interpretation depends on the filter but it's usually the file descriptor. In the constructor ident can either be an int or an object with a `fileno()` method. kevent stores the integer internally.

`kevent.filter`

Name of the kernel filter.

Constante	Signification
`KQ_FILTER_READ`	Takes a descriptor and returns whenever there is data available to read
`KQ_FILTER_WRITE`	Takes a descriptor and returns whenever there is data available to write
`KQ_FILTER_AIO`	AIO requests
`KQ_FILTER_VNODE`	Returns when one or more of the requested events watched in *fflag* occurs
`KQ_FILTER_PROC`	Watch for events on a process id
`KQ_FILTER_NETDEV`	Watch for events on a network device [not available on macOS]
`KQ_FILTER_SIGNAL`	Returns whenever the watched signal is delivered to the process
`KQ_FILTER_TIMER`	Establishes an arbitrary timer

`kevent.flags`

Filter action.

Constante	Signification
KQ_EV_ADD	Adds or modifies an event
KQ_EV_DELETE	Removes an event from the queue
KQ_EV_ENABLE	Permitscontrol() to returns the event
KQ_EV_DISABLE	Disablesevent
KQ_EV_ONESHOT	Removes event after first occurrence
KQ_EV_CLEAR	Reset the state after an event is retrieved
KQ_EV_SYSFLAGS	internal event
KQ_EV_FLAG1	internal event
KQ_EV_EOF	Filter specific EOF condition
KQ_EV_ERROR	See return values

kevent.**fflags**

Filter specific flags.

KQ_FILTER_READ and KQ_FILTER_WRITE filter flags :

Constante	Signification
KQ_NOTE_LOWAT	low water mark of a socket buffer

KQ_FILTER_VNODE filter flags :

Constante	Signification
KQ_NOTE_DELETE	*unlink()* was called
KQ_NOTE_WRITE	a write occurred
KQ_NOTE_EXTEND	the file was extended
KQ_NOTE_ATTRIB	an attribute was changed
KQ_NOTE_LINK	the link count has changed
KQ_NOTE_RENAME	the file was renamed
KQ_NOTE_REVOKE	access to the file was revoked

KQ_FILTER_PROC filter flags :

Constante	Signification
KQ_NOTE_EXIT	the process has exited
KQ_NOTE_FORK	the process has called *fork()*
KQ_NOTE_EXEC	the process has executed a new process
KQ_NOTE_PCTRLMASK	internal filter flag
KQ_NOTE_PDATAMASK	internal filter flag
KQ_NOTE_TRACK	follow a process across *fork()*
KQ_NOTE_CHILD	returned on the child process for *NOTE_TRACK*
KQ_NOTE_TRACKERR	unable to attach to a child

KQ_FILTER_NETDEV filter flags (not available on macOS) :

Constante	Signification
KQ_NOTE_LINKUP	link is up
KQ_NOTE_LINKDOWN	link is down
KQ_NOTE_LINKINV	link state is invalid

kevent.**data**

Filter specific data.

`kevent.udata`

User defined value.

19.5 `selectors` --- High-level I/O multiplexing

Ajouté dans la version 3.4.

Source code : Lib/selectors.py

19.5.1 Introduction

This module allows high-level and efficient I/O multiplexing, built upon the `select` module primitives. Users are encouraged to use this module instead, unless they want precise control over the OS-level primitives used.

It defines a `BaseSelector` abstract base class, along with several concrete implementations (`KqueueSelector`, `EpollSelector`...), that can be used to wait for I/O readiness notification on multiple file objects. In the following, "file object" refers to any object with a `fileno()` method, or a raw file descriptor. See *file object*.

`DefaultSelector` is an alias to the most efficient implementation available on the current platform : this should be the default choice for most users.

Note

The type of file objects supported depends on the platform : on Windows, sockets are supported, but not pipes, whereas on Unix, both are supported (some other types may be supported as well, such as fifos or special file devices).

Voir aussi

select

Low-level I/O multiplexing module.

Availability : not WASI.

This module does not work or is not available on WebAssembly. See *Plateformes WebAssembly* for more information.

19.5.2 Classes

Classes hierarchy :

```
BaseSelector
+-- SelectSelector
+-- PollSelector
+-- EpollSelector
+-- DevpollSelector
+-- KqueueSelector
```

In the following, *events* is a bitwise mask indicating which I/O events should be waited for on a given file object. It can be a combination of the modules constants below :

Constante	Signification
`selectors.EVENT_READ`	Available for read
`selectors.EVENT_WRITE`	Available for write

class `selectors.SelectorKey`

A `SelectorKey` is a `namedtuple` used to associate a file object to its underlying file descriptor, selected event mask and attached data. It is returned by several `BaseSelector` methods.

fileobj

File object registered.

fd

Underlying file descriptor.

events

Events that must be waited for on this file object.

data

Optional opaque data associated to this file object : for example, this could be used to store a per-client session ID.

class `selectors.BaseSelector`

A `BaseSelector` is used to wait for I/O event readiness on multiple file objects. It supports file stream registration, unregistration, and a method to wait for I/O events on those streams, with an optional timeout. It's an abstract base class, so cannot be instantiated. Use `DefaultSelector` instead, or one of `SelectSelector`, `KqueueSelector` etc. if you want to specifically use an implementation, and your platform supports it. `BaseSelector` and its concrete implementations support the *context manager* protocol.

abstractmethod register(*fileobj*, *events*, *data=None*)

Register a file object for selection, monitoring it for I/O events.

fileobj is the file object to monitor. It may either be an integer file descriptor or an object with a `fileno()` method. *events* is a bitwise mask of events to monitor. *data* is an opaque object.

This returns a new `SelectorKey` instance, or raises a `ValueError` in case of invalid event mask or file descriptor, or `KeyError` if the file object is already registered.

abstractmethod unregister(*fileobj*)

Unregister a file object from selection, removing it from monitoring. A file object shall be unregistered prior to being closed.

fileobj must be a file object previously registered.

This returns the associated `SelectorKey` instance, or raises a `KeyError` if *fileobj* is not registered. It will raise `ValueError` if *fileobj* is invalid (e.g. it has no `fileno()` method or its `fileno()` method has an invalid return value).

modify(*fileobj*, *events*, *data=None*)

Change a registered file object's monitored events or attached data.

This is equivalent to `BaseSelector.unregister(fileobj)` followed by `BaseSelector.register(fileobj, events, data)`, except that it can be implemented more efficiently.

This returns a new `SelectorKey` instance, or raises a `ValueError` in case of invalid event mask or file descriptor, or `KeyError` if the file object is not registered.

abstractmethod select(*timeout=None*)

Wait until some registered file objects become ready, or the timeout expires.

If `timeout > 0`, this specifies the maximum wait time, in seconds. If `timeout <= 0`, the call won't block, and will report the currently ready file objects. If *timeout* is `None`, the call will block until a monitored file object becomes ready.

This returns a list of `(key, events)` tuples, one for each ready file object.

key is the `SelectorKey` instance corresponding to a ready file object. *events* is a bitmask of events ready on this file object.

> **Note**
>
> This method can return before any file object becomes ready or the timeout has elapsed if the current process receives a signal : in this case, an empty list will be returned.

Modifié dans la version 3.5 : The selector is now retried with a recomputed timeout when interrupted by a signal if the signal handler did not raise an exception (see **PEP 475** for the rationale), instead of returning an empty list of events before the timeout.

close()

Close the selector.

This must be called to make sure that any underlying resource is freed. The selector shall not be used once it has been closed.

get_key(*fileobj*)

Return the key associated with a registered file object.

This returns the `SelectorKey` instance associated to this file object, or raises `KeyError` if the file object is not registered.

abstractmethod get_map()

Return a mapping of file objects to selector keys.

This returns a `Mapping` instance mapping registered file objects to their associated `SelectorKey` instance.

class selectors.**DefaultSelector**

The default selector class, using the most efficient implementation available on the current platform. This should be the default choice for most users.

class selectors.**SelectSelector**

`select.select()`-based selector.

class selectors.**PollSelector**

`select.poll()`-based selector.

class selectors.**EpollSelector**

`select.epoll()`-based selector.

fileno()

This returns the file descriptor used by the underlying `select.epoll()` object.

class selectors.**DevpollSelector**

`select.devpoll()`-based selector.

fileno()

This returns the file descriptor used by the underlying `select.devpoll()` object.

Ajouté dans la version 3.5.

class selectors.**KqueueSelector**

`select.kqueue()`-based selector.

fileno()

This returns the file descriptor used by the underlying `select.kqueue()` object.

19.5.3 Exemples

Here is a simple echo server implementation :

```
import selectors
import socket

sel = selectors.DefaultSelector()

def accept(sock, mask):
    conn, addr = sock.accept()  # Should be ready
    print('accepted', conn, 'from', addr)
    conn.setblocking(False)
    sel.register(conn, selectors.EVENT_READ, read)

def read(conn, mask):
    data = conn.recv(1000)  # Should be ready
    if data:
        print('echoing', repr(data), 'to', conn)
        conn.send(data)  # Hope it won't block
    else:
        print('closing', conn)
        sel.unregister(conn)
        conn.close()

sock = socket.socket()
sock.bind(('localhost', 1234))
sock.listen(100)
sock.setblocking(False)
sel.register(sock, selectors.EVENT_READ, accept)

while True:
    events = sel.select()
    for key, mask in events:
        callback = key.data
        callback(key.fileobj, mask)
```

19.6 `signal` --- Set handlers for asynchronous events

Source code : Lib/signal.py

This module provides mechanisms to use signal handlers in Python.

19.6.1 General rules

The `signal.signal()` function allows defining custom handlers to be executed when a signal is received. A small number of default handlers are installed : `SIGPIPE` is ignored (so write errors on pipes and sockets can be reported as ordinary Python exceptions) and `SIGINT` is translated into a `KeyboardInterrupt` exception if the parent process has not changed it.

A handler for a particular signal, once set, remains installed until it is explicitly reset (Python emulates the BSD style interface regardless of the underlying implementation), with the exception of the handler for `SIGCHLD`, which follows the underlying implementation.

On WebAssembly platforms, signals are emulated and therefore behave differently. Several functions and signals are not available on these platforms.

Execution of Python signal handlers

A Python signal handler does not get executed inside the low-level (C) signal handler. Instead, the low-level signal handler sets a flag which tells the *virtual machine* to execute the corresponding Python signal handler at a later point(for example at the next *bytecode* instruction). This has consequences :

— It makes little sense to catch synchronous errors like `SIGFPE` or `SIGSEGV` that are caused by an invalid operation in C code. Python will return from the signal handler to the C code, which is likely to raise the same signal again, causing Python to apparently hang. From Python 3.3 onwards, you can use the `faulthandler` module to report on synchronous errors.

— A long-running calculation implemented purely in C (such as regular expression matching on a large body of text) may run uninterrupted for an arbitrary amount of time, regardless of any signals received. The Python signal handlers will be called when the calculation finishes.

— If the handler raises an exception, it will be raised "out of thin air" in the main thread. See the *note below* for a discussion.

Signals and threads

Python signal handlers are always executed in the main Python thread of the main interpreter, even if the signal was received in another thread. This means that signals can't be used as a means of inter-thread communication. You can use the synchronization primitives from the `threading` module instead.

Besides, only the main thread of the main interpreter is allowed to set a new signal handler.

19.6.2 Module contents

Modifié dans la version 3.5 : signal (SIG*), handler (`SIG_DFL`, `SIG_IGN`) and sigmask (`SIG_BLOCK`, `SIG_UNBLOCK`, `SIG_SETMASK`) related constants listed below were turned into `enums` (`Signals`, `Handlers` and `Sigmasks` respectively). `getsignal()`, `pthread_sigmask()`, `sigpending()` and `sigwait()` functions return human-readable `enums` as `Signals` objects.

The signal module defines three enums :

class signal.**Signals**

`enum.IntEnum` collection of SIG* constants and the CTRL_* constants.

Ajouté dans la version 3.5.

class signal.**Handlers**

`enum.IntEnum` collection the constants `SIG_DFL` and `SIG_IGN`.

Ajouté dans la version 3.5.

class signal.**Sigmasks**

`enum.IntEnum` collection the constants `SIG_BLOCK`, `SIG_UNBLOCK` and `SIG_SETMASK`.

Availability : Unix.

See the man page `sigprocmask(2)` and `pthread_sigmask(3)` for further information.

Ajouté dans la version 3.5.

The variables defined in the `signal` module are :

signal.**SIG_DFL**

This is one of two standard signal handling options ; it will simply perform the default function for the signal. For example, on most systems the default action for `SIGQUIT` is to dump core and exit, while the default action for `SIGCHLD` is to simply ignore it.

signal.**SIG_IGN**

This is another standard signal handler, which will simply ignore the given signal.

signal.**SIGABRT**

Abort signal from *abort(3)*.

signal.**SIGALRM**

Timer signal from *alarm(2)*.

Availability : Unix.

signal.**SIGBREAK**

Interrupt from keyboard (CTRL + BREAK).

Availability : Windows.

signal.**SIGBUS**

Bus error (bad memory access).

Availability : Unix.

signal.**SIGCHLD**

Child process stopped or terminated.

Availability : Unix.

signal.**SIGCLD**

Alias to *SIGCHLD*.

Availability : not macOS.

signal.**SIGCONT**

Continue the process if it is currently stopped

Availability : Unix.

signal.**SIGFPE**

Floating-point exception. For example, division by zero.

> **Voir aussi**
>
> *ZeroDivisionError* is raised when the second argument of a division or modulo operation is zero.

signal.**SIGHUP**

Hangup detected on controlling terminal or death of controlling process.

Availability : Unix.

signal.**SIGILL**

Illegal instruction.

signal.**SIGINT**

Interrupt from keyboard (CTRL + C).

Default action is to raise *KeyboardInterrupt*.

signal.**SIGKILL**

Kill signal.

It cannot be caught, blocked, or ignored.

Availability : Unix.

signal.**SIGPIPE**

Broken pipe : write to pipe with no readers.

Default action is to ignore the signal.

Availability : Unix.

signal.**SIGSEGV**

Segmentation fault : invalid memory reference.

signal.**SIGSTKFLT**

Stack fault on coprocessor. The Linux kernel does not raise this signal : it can only be raised in user space.

Availability : Linux.

On architectures where the signal is available. See the man page *signal(7)* for further information.

Ajouté dans la version 3.11.

signal.**SIGTERM**

Termination signal.

signal.**SIGUSR1**

User-defined signal 1.

Availability : Unix.

signal.**SIGUSR2**

User-defined signal 2.

Availability : Unix.

signal.**SIGWINCH**

Window resize signal.

Availability : Unix.

SIG*

All the signal numbers are defined symbolically. For example, the hangup signal is defined as *signal.SIGHUP*; the variable names are identical to the names used in C programs, as found in `<signal.h>`. The Unix man page for '`signal()`' lists the existing signals (on some systems this is *signal(2)*, on others the list is in *signal(7)*). Note that not all systems define the same set of signal names; only those names defined by the system are defined by this module.

signal.**CTRL_C_EVENT**

The signal corresponding to the `Ctrl+C` keystroke event. This signal can only be used with *os.kill()*.

Availability : Windows.

Ajouté dans la version 3.2.

signal.**CTRL_BREAK_EVENT**

The signal corresponding to the `Ctrl+Break` keystroke event. This signal can only be used with *os.kill()*.

Availability : Windows.

Ajouté dans la version 3.2.

signal.**NSIG**

One more than the number of the highest signal number. Use *valid_signals()* to get valid signal numbers.

signal.**ITIMER_REAL**

Decrements interval timer in real time, and delivers *SIGALRM* upon expiration.

signal.**ITIMER_VIRTUAL**

Decrements interval timer only when the process is executing, and delivers SIGVTALRM upon expiration.

signal.**ITIMER_PROF**

Decrements interval timer both when the process executes and when the system is executing on behalf of the process. Coupled with ITIMER_VIRTUAL, this timer is usually used to profile the time spent by the application in user and kernel space. SIGPROF is delivered upon expiration.

signal.**SIG_BLOCK**

A possible value for the *how* parameter to `pthread_sigmask()` indicating that signals are to be blocked.

Ajouté dans la version 3.3.

signal.**SIG_UNBLOCK**

A possible value for the *how* parameter to `pthread_sigmask()` indicating that signals are to be unblocked.

Ajouté dans la version 3.3.

signal.**SIG_SETMASK**

A possible value for the *how* parameter to `pthread_sigmask()` indicating that the signal mask is to be replaced.

Ajouté dans la version 3.3.

The `signal` module defines one exception :

exception signal.**ItimerError**

Raised to signal an error from the underlying `setitimer()` or `getitimer()` implementation. Expect this error if an invalid interval timer or a negative time is passed to `setitimer()`. This error is a subtype of `OSError`.

Ajouté dans la version 3.3 : This error used to be a subtype of `IOError`, which is now an alias of `OSError`.

The `signal` module defines the following functions :

signal.**alarm**(*time*)

If *time* is non-zero, this function requests that a `SIGALRM` signal be sent to the process in *time* seconds. Any previously scheduled alarm is canceled (only one alarm can be scheduled at any time). The returned value is then the number of seconds before any previously set alarm was to have been delivered. If *time* is zero, no alarm is scheduled, and any scheduled alarm is canceled. If the return value is zero, no alarm is currently scheduled.

Availability : Unix.

See the man page `alarm(2)` for further information.

signal.**getsignal**(*signalnum*)

Return the current signal handler for the signal *signalnum*. The returned value may be a callable Python object, or one of the special values `signal.SIG_IGN`, `signal.SIG_DFL` or `None`. Here, `signal.SIG_IGN` means that the signal was previously ignored, `signal.SIG_DFL` means that the default way of handling the signal was previously in use, and `None` means that the previous signal handler was not installed from Python.

signal.**strsignal**(*signalnum*)

Returns the description of signal *signalnum*, such as "Interrupt" for `SIGINT`. Returns `None` if *signalnum* has no description. Raises `ValueError` if *signalnum* is invalid.

Ajouté dans la version 3.8.

signal.**valid_signals**()

Return the set of valid signal numbers on this platform. This can be less than `range(1, NSIG)` if some signals are reserved by the system for internal use.

Ajouté dans la version 3.8.

signal.**pause**()

Cause the process to sleep until a signal is received ; the appropriate handler will then be called. Returns nothing.

Availability : Unix.

See the man page `signal(2)` for further information.

See also `sigwait()`, `sigwaitinfo()`, `sigtimedwait()` and `sigpending()`.

signal.**raise_signal**(*signum*)

Sends a signal to the calling process. Returns nothing.

Ajouté dans la version 3.8.

signal.**pidfd_send_signal**(*pidfd*, *sig*, *siginfo=None*, *flags=0*)

Send signal *sig* to the process referred to by file descriptor *pidfd*. Python does not currently support the *siginfo* parameter ; it must be `None`. The *flags* argument is provided for future extensions ; no flag values are currently defined.

See the `pidfd_send_signal(2)` man page for more information.

Availability : Linux >= 5.1, Android >= `build-time` API level 31

Ajouté dans la version 3.9.

signal.**pthread_kill**(*thread_id*, *signalnum*)

Send the signal *signalnum* to the thread *thread_id*, another thread in the same process as the caller. The target thread can be executing any code (Python or not). However, if the target thread is executing the Python interpreter, the Python signal handlers will be *executed by the main thread of the main interpreter*. Therefore, the only point of sending a signal to a particular Python thread would be to force a running system call to fail with `InterruptedError`.

Use `threading.get_ident()` or the `ident` attribute of `threading.Thread` objects to get a suitable value for *thread_id*.

If *signalnum* is 0, then no signal is sent, but error checking is still performed ; this can be used to check if the target thread is still running.

Raises an *auditing event* `signal.pthread_kill` with arguments `thread_id`, `signalnum`.

Availability : Unix.

See the man page `pthread_kill(3)` for further information.

See also `os.kill()`.

Ajouté dans la version 3.3.

signal.**pthread_sigmask**(*how*, *mask*)

Fetch and/or change the signal mask of the calling thread. The signal mask is the set of signals whose delivery is currently blocked for the caller. Return the old signal mask as a set of signals.

The behavior of the call is dependent on the value of *how*, as follows.

— `SIG_BLOCK` : The set of blocked signals is the union of the current set and the *mask* argument.
— `SIG_UNBLOCK` : The signals in *mask* are removed from the current set of blocked signals. It is permissible to attempt to unblock a signal which is not blocked.
— `SIG_SETMASK` : The set of blocked signals is set to the *mask* argument.

mask is a set of signal numbers (e.g. {`signal.SIGINT`, `signal.SIGTERM`}). Use `valid_signals()` for a full mask including all signals.

For example, `signal.pthread_sigmask(signal.SIG_BLOCK, [])` reads the signal mask of the calling thread.

`SIGKILL` and `SIGSTOP` cannot be blocked.

Availability : Unix.

See the man page `sigprocmask(2)` and `pthread_sigmask(3)` for further information.

See also `pause()`, `sigpending()` and `sigwait()`.

Ajouté dans la version 3.3.

signal.**setitimer**(*which*, *seconds*, *interval=0.0*)

Sets given interval timer (one of `signal.ITIMER_REAL`, `signal.ITIMER_VIRTUAL` or `signal.ITIMER_PROF`) specified by *which* to fire after *seconds* (float is accepted, different from `alarm()`) and after that every *interval* seconds (if *interval* is non-zero). The interval timer specified by *which* can be cleared by setting *seconds* to zero.

When an interval timer fires, a signal is sent to the process. The signal sent is dependent on the timer being used ; `signal.ITIMER_REAL` will deliver `SIGALRM`, `signal.ITIMER_VIRTUAL` sends `SIGVTALRM`, and `signal.ITIMER_PROF` will deliver `SIGPROF`.

The old values are returned as a tuple : (delay, interval).

Attempting to pass an invalid interval timer will cause an `ItimerError`.

Availability : Unix.

`signal.getitimer(which)`

Returns current value of a given interval timer specified by *which*.

Availability : Unix.

`signal.set_wakeup_fd(fd, *, warn_on_full_buffer=True)`

Set the wakeup file descriptor to *fd*. When a signal is received, the signal number is written as a single byte into the fd. This can be used by a library to wakeup a poll or select call, allowing the signal to be fully processed.

The old wakeup fd is returned (or -1 if file descriptor wakeup was not enabled). If *fd* is -1, file descriptor wakeup is disabled. If not -1, *fd* must be non-blocking. It is up to the library to remove any bytes from *fd* before calling poll or select again.

When threads are enabled, this function can only be called from *the main thread of the main interpreter* ; attempting to call it from other threads will cause a `ValueError` exception to be raised.

There are two common ways to use this function. In both approaches, you use the fd to wake up when a signal arrives, but then they differ in how they determine *which* signal or signals have arrived.

In the first approach, we read the data out of the fd's buffer, and the byte values give you the signal numbers. This is simple, but in rare cases it can run into a problem : generally the fd will have a limited amount of buffer space, and if too many signals arrive too quickly, then the buffer may become full, and some signals may be lost. If you use this approach, then you should set `warn_on_full_buffer=True`, which will at least cause a warning to be printed to stderr when signals are lost.

In the second approach, we use the wakeup fd *only* for wakeups, and ignore the actual byte values. In this case, all we care about is whether the fd's buffer is empty or non-empty ; a full buffer doesn't indicate a problem at all. If you use this approach, then you should set `warn_on_full_buffer=False`, so that your users are not confused by spurious warning messages.

Modifié dans la version 3.5 : On Windows, the function now also supports socket handles.

Modifié dans la version 3.7 : Added `warn_on_full_buffer` parameter.

`signal.siginterrupt(signalnum, flag)`

Change system call restart behaviour : if *flag* is `False`, system calls will be restarted when interrupted by signal *signalnum*, otherwise system calls will be interrupted. Returns nothing.

Availability : Unix.

See the man page `siginterrupt(3)` for further information.

Note that installing a signal handler with `signal()` will reset the restart behaviour to interruptible by implicitly calling `siginterrupt()` with a true *flag* value for the given signal.

`signal.signal(signalnum, handler)`

Set the handler for signal *signalnum* to the function *handler*. *handler* can be a callable Python object taking two arguments (see below), or one of the special values `signal.SIG_IGN` or `signal.SIG_DFL`. The previous signal handler will be returned (see the description of `getsignal()` above). (See the Unix man page `signal(2)` for further information.)

When threads are enabled, this function can only be called from *the main thread of the main interpreter* ; attempting to call it from other threads will cause a `ValueError` exception to be raised.

The *handler* is called with two arguments : the signal number and the current stack frame (`None` or a frame object ; for a description of frame objects, see the description in the type hierarchy or see the attribute descriptions in the `inspect` module).

On Windows, `signal()` can only be called with `SIGABRT`, `SIGFPE`, `SIGILL`, `SIGINT`, `SIGSEGV`, `SIGTERM`, or `SIGBREAK`. A `ValueError` will be raised in any other case. Note that not all systems define the same set of signal names ; an `AttributeError` will be raised if a signal name is not defined as `SIG*` module level

constant.

signal.**sigpending**()

Examine the set of signals that are pending for delivery to the calling thread (i.e., the signals which have been raised while blocked). Return the set of the pending signals.

Availability : Unix.

See the man page `sigpending(2)` for further information.

See also `pause()`, `pthread_sigmask()` and `sigwait()`.

Ajouté dans la version 3.3.

signal.**sigwait**(*sigset*)

Suspend execution of the calling thread until the delivery of one of the signals specified in the signal set *sigset*. The function accepts the signal (removes it from the pending list of signals), and returns the signal number.

Availability : Unix.

See the man page `sigwait(3)` for further information.

See also `pause()`, `pthread_sigmask()`, `sigpending()`, `sigwaitinfo()` and `sigtimedwait()`.

Ajouté dans la version 3.3.

signal.**sigwaitinfo**(*sigset*)

Suspend execution of the calling thread until the delivery of one of the signals specified in the signal set *sigset*. The function accepts the signal and removes it from the pending list of signals. If one of the signals in *sigset* is already pending for the calling thread, the function will return immediately with information about that signal. The signal handler is not called for the delivered signal. The function raises an `InterruptedError` if it is interrupted by a signal that is not in *sigset*.

The return value is an object representing the data contained in the `siginfo_t` structure, namely : `si_signo`, `si_code`, `si_errno`, `si_pid`, `si_uid`, `si_status`, `si_band`.

Availability : Unix.

See the man page `sigwaitinfo(2)` for further information.

See also `pause()`, `sigwait()` and `sigtimedwait()`.

Ajouté dans la version 3.3.

Modifié dans la version 3.5 : The function is now retried if interrupted by a signal not in *sigset* and the signal handler does not raise an exception (see **PEP 475** for the rationale).

signal.**sigtimedwait**(*sigset*, *timeout*)

Like `sigwaitinfo()`, but takes an additional *timeout* argument specifying a timeout. If *timeout* is specified as `0`, a poll is performed. Returns `None` if a timeout occurs.

Availability : Unix.

See the man page `sigtimedwait(2)` for further information.

See also `pause()`, `sigwait()` and `sigwaitinfo()`.

Ajouté dans la version 3.3.

Modifié dans la version 3.5 : The function is now retried with the recomputed *timeout* if interrupted by a signal not in *sigset* and the signal handler does not raise an exception (see **PEP 475** for the rationale).

19.6.3 Examples

Here is a minimal example program. It uses the `alarm()` function to limit the time spent waiting to open a file ; this is useful if the file is for a serial device that may not be turned on, which would normally cause the `os.open()` to hang indefinitely. The solution is to set a 5-second alarm before opening the file ; if the operation takes too long, the alarm signal will be sent, and the handler raises an exception.

```
import signal, os

def handler(signum, frame):
    signame = signal.Signals(signum).name
    print(f'Signal handler called with signal {signame} ({signum})')
    raise OSError("Couldn't open device!")

# Set the signal handler and a 5-second alarm
signal.signal(signal.SIGALRM, handler)
signal.alarm(5)

# This open() may hang indefinitely
fd = os.open('/dev/ttyS0', os.O_RDWR)

signal.alarm(0)          # Disable the alarm
```

19.6.4 Note on SIGPIPE

Piping output of your program to tools like *head(1)* will cause a *SIGPIPE* signal to be sent to your process when the receiver of its standard output closes early. This results in an exception like `BrokenPipeError: [Errno 32] Broken pipe`. To handle this case, wrap your entry point to catch this exception as follows :

```
import os
import sys

def main():
    try:
        # simulate large output (your code replaces this loop)
        for x in range(10000):
            print("y")
        # flush output here to force SIGPIPE to be triggered
        # while inside this try block.
        sys.stdout.flush()
    except BrokenPipeError:
        # Python flushes standard streams on exit; redirect remaining output
        # to devnull to avoid another BrokenPipeError at shutdown
        devnull = os.open(os.devnull, os.O_WRONLY)
        os.dup2(devnull, sys.stdout.fileno())
        sys.exit(1)  # Python exits with error code 1 on EPIPE

if __name__ == '__main__':
    main()
```

Do not set *SIGPIPE*'s disposition to *SIG_DFL* in order to avoid *BrokenPipeError*. Doing that would cause your program to exit unexpectedly whenever any socket connection is interrupted while your program is still writing to it.

19.6.5 Note on Signal Handlers and Exceptions

If a signal handler raises an exception, the exception will be propagated to the main thread and may be raised after any *bytecode* instruction. Most notably, a *KeyboardInterrupt* may appear at any point during execution. Most Python code, including the standard library, cannot be made robust against this, and so a *KeyboardInterrupt* (or any other exception resulting from a signal handler) may on rare occasions put the program in an unexpected state.

To illustrate this issue, consider the following code :

```
class SpamContext:
    def __init__(self):
```

(suite sur la page suivante)

(suite de la page précédente)

```
        self.lock = threading.Lock()

    def __enter__(self):
        # If KeyboardInterrupt occurs here, everything is fine
        self.lock.acquire()
        # If KeyboardInterrupt occurs here, __exit__ will not be called
        ...
        # KeyboardInterrupt could occur just before the function returns

    def __exit__(self, exc_type, exc_val, exc_tb):
        ...
        self.lock.release()
```

For many programs, especially those that merely want to exit on `KeyboardInterrupt`, this is not a problem, but applications that are complex or require high reliability should avoid raising exceptions from signal handlers. They should also avoid catching `KeyboardInterrupt` as a means of gracefully shutting down. Instead, they should install their own `SIGINT` handler. Below is an example of an HTTP server that avoids `KeyboardInterrupt` :

```
import signal
import socket
from selectors import DefaultSelector, EVENT_READ
from http.server import HTTPServer, SimpleHTTPRequestHandler

interrupt_read, interrupt_write = socket.socketpair()

def handler(signum, frame):
    print('Signal handler called with signal', signum)
    interrupt_write.send(b'\0')
signal.signal(signal.SIGINT, handler)

def serve_forever(httpd):
    sel = DefaultSelector()
    sel.register(interrupt_read, EVENT_READ)
    sel.register(httpd, EVENT_READ)

    while True:
        for key, _ in sel.select():
            if key.fileobj == interrupt_read:
                interrupt_read.recv(1)
                return
            if key.fileobj == httpd:
                httpd.handle_request()

print("Serving on port 8000")
httpd = HTTPServer(('', 8000), SimpleHTTPRequestHandler)
serve_forever(httpd)
print("Shutdown...")
```

19.7 `mmap` --- Memory-mapped file support

Availability : not WASI.

This module does not work or is not available on WebAssembly. See *Plateformes WebAssembly* for more information.

Memory-mapped file objects behave like both `bytearray` and like *file objects*. You can use mmap objects in most places where `bytearray` are expected ; for example, you can use the `re` module to search through a memory-mapped

file. You can also change a single byte by doing `obj[index] = 97`, or change a subsequence by assigning to a slice : `obj[i1:i2] = b'...'`. You can also read and write data starting at the current file position, and `seek()` through the file to different positions.

A memory-mapped file is created by the `mmap` constructor, which is different on Unix and on Windows. In either case you must provide a file descriptor for a file opened for update. If you wish to map an existing Python file object, use its `fileno()` method to obtain the correct value for the *fileno* parameter. Otherwise, you can open the file using the `os.open()` function, which returns a file descriptor directly (the file still needs to be closed when done).

> **Note**
>
> If you want to create a memory-mapping for a writable, buffered file, you should `flush()` the file first. This is necessary to ensure that local modifications to the buffers are actually available to the mapping.

For both the Unix and Windows versions of the constructor, *access* may be specified as an optional keyword parameter. *access* accepts one of four values : `ACCESS_READ`, `ACCESS_WRITE`, or `ACCESS_COPY` to specify read-only, write-through or copy-on-write memory respectively, or `ACCESS_DEFAULT` to defer to *prot*. *access* can be used on both Unix and Windows. If *access* is not specified, Windows mmap returns a write-through mapping. The initial memory values for all three access types are taken from the specified file. Assignment to an `ACCESS_READ` memory map raises a `TypeError` exception. Assignment to an `ACCESS_WRITE` memory map affects both memory and the underlying file. Assignment to an `ACCESS_COPY` memory map affects memory but does not update the underlying file.

Modifié dans la version 3.7 : Added `ACCESS_DEFAULT` constant.

To map anonymous memory, -1 should be passed as the fileno along with the length.

class `mmap.`**`mmap`**(*fileno*, *length*, *tagname=None*, *access=ACCESS_DEFAULT*, *offset=0*)

(Windows version) Maps *length* bytes from the file specified by the file handle *fileno*, and creates a mmap object. If *length* is larger than the current size of the file, the file is extended to contain *length* bytes. If *length* is `0`, the maximum length of the map is the current size of the file, except that if the file is empty Windows raises an exception (you cannot create an empty mapping on Windows).

tagname, if specified and not `None`, is a string giving a tag name for the mapping. Windows allows you to have many different mappings against the same file. If you specify the name of an existing tag, that tag is opened, otherwise a new tag of this name is created. If this parameter is omitted or `None`, the mapping is created without a name. Avoiding the use of the *tagname* parameter will assist in keeping your code portable between Unix and Windows.

offset may be specified as a non-negative integer offset. mmap references will be relative to the offset from the beginning of the file. *offset* defaults to 0. *offset* must be a multiple of the `ALLOCATIONGRANULARITY`.

Raises an *auditing event* `mmap.__new__` with arguments `fileno`, `length`, `access`, `offset`.

class `mmap.`**`mmap`**(*fileno*, *length*, *flags=MAP_SHARED*, *prot=PROT_WRITE | PROT_READ*, *access=ACCESS_DEFAULT*, *offset=0*, *, *trackfd=True*)

(Unix version) Maps *length* bytes from the file specified by the file descriptor *fileno*, and returns a mmap object. If *length* is `0`, the maximum length of the map will be the current size of the file when `mmap` is called.

flags specifies the nature of the mapping. `MAP_PRIVATE` creates a private copy-on-write mapping, so changes to the contents of the mmap object will be private to this process, and `MAP_SHARED` creates a mapping that's shared with all other processes mapping the same areas of the file. The default value is `MAP_SHARED`. Some systems have additional possible flags with the full list specified in *MAP_* constants*.

prot, if specified, gives the desired memory protection; the two most useful values are `PROT_READ` and `PROT_WRITE`, to specify that the pages may be read or written. *prot* defaults to `PROT_READ | PROT_WRITE`.

access may be specified in lieu of *flags* and *prot* as an optional keyword parameter. It is an error to specify both *flags*, *prot* and *access*. See the description of *access* above for information on how to use this parameter.

offset may be specified as a non-negative integer offset. mmap references will be relative to the offset from the beginning of the file. *offset* defaults to 0. *offset* must be a multiple of `ALLOCATIONGRANULARITY` which is equal to `PAGESIZE` on Unix systems.

If *trackfd* is `False`, the file descriptor specified by *fileno* will not be duplicated, and the resulting `mmap` object will not be associated with the map's underlying file. This means that the `size()` and `resize()` methods will fail. This mode is useful to limit the number of open file descriptors.

To ensure validity of the created memory mapping the file specified by the descriptor *fileno* is internally automatically synchronized with the physical backing store on macOS.

Modifié dans la version 3.13 : The *trackfd* parameter was added.

This example shows a simple way of using `mmap` :

```
import mmap

# write a simple example file
with open("hello.txt", "wb") as f:
    f.write(b"Hello Python!\n")

with open("hello.txt", "r+b") as f:
    # memory-map the file, size 0 means whole file
    mm = mmap.mmap(f.fileno(), 0)
    # read content via standard file methods
    print(mm.readline())  # prints b"Hello Python!\n"
    # read content via slice notation
    print(mm[:5])  # prints b"Hello"
    # update content using slice notation;
    # note that new content must have same size
    mm[6:] = b" world!\n"
    # ... and read again using standard file methods
    mm.seek(0)
    print(mm.readline())  # prints b"Hello  world!\n"
    # close the map
    mm.close()
```

`mmap` can also be used as a context manager in a `with` statement :

```
import mmap

with mmap.mmap(-1, 13) as mm:
    mm.write(b"Hello world!")
```

Ajouté dans la version 3.2 : Context manager support.

The next example demonstrates how to create an anonymous map and exchange data between the parent and child processes :

```
import mmap
import os

mm = mmap.mmap(-1, 13)
mm.write(b"Hello world!")

pid = os.fork()

if pid == 0:  # In a child process
    mm.seek(0)
    print(mm.readline())

    mm.close()
```

Raises an *auditing event* `mmap.__new__` with arguments `fileno`, `length`, `access`, `offset`.

Memory-mapped file objects support the following methods :

close()

Closes the mmap. Subsequent calls to other methods of the object will result in a ValueError exception being raised. This will not close the open file.

closed

`True` if the file is closed.

Ajouté dans la version 3.2.

find(*sub*[, *start*[, *end*]])

Returns the lowest index in the object where the subsequence *sub* is found, such that *sub* is contained in the range [*start*, *end*]. Optional arguments *start* and *end* are interpreted as in slice notation. Returns `-1` on failure.

Modifié dans la version 3.5 : N'importe quel *bytes-like object* est maintenant accepté.

flush([*offset*[, *size*]])

Flushes changes made to the in-memory copy of a file back to disk. Without use of this call there is no guarantee that changes are written back before the object is destroyed. If *offset* and *size* are specified, only changes to the given range of bytes will be flushed to disk ; otherwise, the whole extent of the mapping is flushed. *offset* must be a multiple of the `PAGESIZE` or `ALLOCATIONGRANULARITY`.

`None` is returned to indicate success. An exception is raised when the call failed.

Modifié dans la version 3.8 : Previously, a nonzero value was returned on success ; zero was returned on error under Windows. A zero value was returned on success ; an exception was raised on error under Unix.

madvise(*option*[, *start*[, *length*]])

Send advice *option* to the kernel about the memory region beginning at *start* and extending *length* bytes. *option* must be one of the *MADV_* constants* available on the system. If *start* and *length* are omitted, the entire mapping is spanned. On some systems (including Linux), *start* must be a multiple of the `PAGESIZE`.

Availability : Systems with the `madvise()` system call.

Ajouté dans la version 3.8.

move(*dest*, *src*, *count*)

Copy the *count* bytes starting at offset *src* to the destination index *dest*. If the mmap was created with `ACCESS_READ`, then calls to move will raise a `TypeError` exception.

read([*n*])

Return a `bytes` containing up to *n* bytes starting from the current file position. If the argument is omitted, `None` or negative, return all bytes from the current file position to the end of the mapping. The file position is updated to point after the bytes that were returned.

Modifié dans la version 3.3 : Argument can be omitted or `None`.

read_byte()

Returns a byte at the current file position as an integer, and advances the file position by 1.

readline()

Returns a single line, starting at the current file position and up to the next newline. The file position is updated to point after the bytes that were returned.

resize(*newsize*)

Resizes the map and the underlying file, if any.

Resizing a map created with *access* of `ACCESS_READ` or `ACCESS_COPY`, will raise a `TypeError` exception. Resizing a map created with with *trackfd* set to `False`, will raise a `ValueError` exception.

On Windows : Resizing the map will raise an `OSError` if there are other maps against the same named file. Resizing an anonymous map (ie against the pagefile) will silently create a new map with the original data copied over up to the length of the new size.

Modifié dans la version 3.11 : Correctly fails if attempting to resize when another map is held Allows resize against an anonymous map on Windows

rfind(*sub*[, *start*[, *end*]])

Returns the highest index in the object where the subsequence *sub* is found, such that *sub* is contained in the range [*start*, *end*]. Optional arguments *start* and *end* are interpreted as in slice notation. Returns `-1` on failure.

Modifié dans la version 3.5 : N'importe quel *bytes-like object* est maintenant accepté.

seek(*pos*[, *whence*])

Set the file's current position. *whence* argument is optional and defaults to `os.SEEK_SET` or `0` (absolute file positioning); other values are `os.SEEK_CUR` or `1` (seek relative to the current position) and `os.SEEK_END` or `2` (seek relative to the file's end).

Modifié dans la version 3.13 : Return the new absolute position instead of `None`.

seekable()

Return whether the file supports seeking, and the return value is always `True`.

Ajouté dans la version 3.13.

size()

Return the length of the file, which can be larger than the size of the memory-mapped area.

tell()

Returns the current position of the file pointer.

write(*bytes*)

Write the bytes in *bytes* into memory at the current position of the file pointer and return the number of bytes written (never less than `len(bytes)`, since if the write fails, a `ValueError` will be raised). The file position is updated to point after the bytes that were written. If the mmap was created with `ACCESS_READ`, then writing to it will raise a `TypeError` exception.

Modifié dans la version 3.5 : N'importe quel *bytes-like object* est maintenant accepté.

Modifié dans la version 3.6 : The number of bytes written is now returned.

write_byte(*byte*)

Write the integer *byte* into memory at the current position of the file pointer; the file position is advanced by `1`. If the mmap was created with `ACCESS_READ`, then writing to it will raise a `TypeError` exception.

19.7.1 MADV_* Constants

mmap.**MADV_NORMAL**

mmap.**MADV_RANDOM**

mmap.**MADV_SEQUENTIAL**

mmap.**MADV_WILLNEED**

mmap.**MADV_DONTNEED**

mmap.**MADV_REMOVE**

mmap.**MADV_DONTFORK**

mmap.**MADV_DOFORK**

mmap.**MADV_HWPOISON**

mmap.**MADV_MERGEABLE**

mmap.**MADV_UNMERGEABLE**

mmap.**MADV_SOFT_OFFLINE**

mmap.**MADV_HUGEPAGE**

mmap.**MADV_NOHUGEPAGE**

mmap.**MADV_DONTDUMP**

mmap.**MADV_DODUMP**

mmap.**MADV_FREE**

mmap.**MADV_NOSYNC**

mmap.**MADV_AUTOSYNC**

mmap.**MADV_NOCORE**

mmap.**MADV_CORE**

mmap.**MADV_PROTECT**

mmap.**MADV_FREE_REUSABLE**

mmap.**MADV_FREE_REUSE**

These options can be passed to `mmap.madvise()`. Not every option will be present on every system.

Availability : Systems with the madvise() system call.

Ajouté dans la version 3.8.

19.7.2 MAP_* Constants

mmap.**MAP_SHARED**

mmap.**MAP_PRIVATE**

mmap.**MAP_32BIT**

mmap.**MAP_ALIGNED_SUPER**

mmap.**MAP_ANON**

mmap.**MAP_ANONYMOUS**

mmap.**MAP_CONCEAL**

mmap.**MAP_DENYWRITE**

mmap.**MAP_EXECUTABLE**

mmap.**MAP_HASSEMAPHORE**

mmap.**MAP_JIT**

mmap.**MAP_NOCACHE**

mmap.**MAP_NOEXTEND**

mmap.**MAP_NORESERVE**

mmap.**MAP_POPULATE**

mmap.**MAP_RESILIENT_CODESIGN**

mmap.**MAP_RESILIENT_MEDIA**

mmap.**MAP_STACK**

mmap.**MAP_TPRO**

mmap.**MAP_TRANSLATED_ALLOW_EXECUTE**

mmap.**MAP_UNIX03**

These are the various flags that can be passed to `mmap.mmap()`. `MAP_ALIGNED_SUPER` is only available at FreeBSD and `MAP_CONCEAL` is only available at OpenBSD. Note that some options might not be present on some systems.

Modifié dans la version 3.10 : Added `MAP_POPULATE` constant.

Ajouté dans la version 3.11 : Added `MAP_STACK` constant.

Ajouté dans la version 3.12 : Added `MAP_ALIGNED_SUPER` and `MAP_CONCEAL` constants.

Ajouté dans la version 3.13 : Added `MAP_32BIT`, `MAP_HASSEMAPHORE`, `MAP_JIT`, `MAP_NOCACHE`, `MAP_NOEXTEND`, `MAP_NORESERVE`, `MAP_RESILIENT_CODESIGN`, `MAP_RESILIENT_MEDIA`, `MAP_TPRO`, `MAP_TRANSLATED_ALLOW_EXECUTE`, and `MAP_UNIX03` constants.

CHAPITRE 20

Traitement des données provenant d'Internet

This chapter describes modules which support handling data formats commonly used on the internet.

20.1 email --- An email and MIME handling package

Code source : Lib/email/__init__.py

The email package is a library for managing email messages. It is specifically *not* designed to do any sending of email messages to SMTP (**RFC 2821**), NNTP, or other servers ; those are functions of modules such as smtplib. The email package attempts to be as RFC-compliant as possible, supporting **RFC 5322** and **RFC 6532**, as well as such MIME-related RFCs as **RFC 2045**, **RFC 2046**, **RFC 2047**, **RFC 2183**, and **RFC 2231**.

Ce paquet peut être divisé entre trois composants majeurs, et un quatrième composant qui contrôle le comportement des trois autres.

Le composant central du paquet est un "modèle d'objet" qui représente les messages. Une application interagit avec le paquet, dans un premier temps, à travers l'interface de modèle d'objet définie dans le sous-module message. L'application peut utiliser cette API pour poser des questions à propos d'un mail existant, pour créer un nouvel e-mail, ou ajouter ou retirer des sous-composants d'e-mail qui utilisent la même interface de modèle d'objet. Suivant la nature des messages et leurs sous-composants MIME, le modèle d'objet d'e-mail est une structure arborescente d'objets qui fournit tout à l'API de EmailMessage.

Les deux autres composants majeurs de ce paquet sont l'analyseur (parser) et le générateur (generator). L'analyseur prend la version sérialisée d'un e-mail (un flux d'octets) et le convertit en une arborescence d'objets EmailMessage. Le générateur prend un objet EmailMessage et le retransforme en un flux d'octets sérialisé (l'analyseur et le générateur gèrent aussi des suites de caractères textuels, mais cette utilisation est déconseillée car il est très facile de finir avec des messages invalides d'une manière ou d'une autre).

Le composant de contrôle est le module policy. Chaque EmailMessage, chaque generator et chaque parser possède un objet associé policy qui contrôle son comportement. Habituellement une application n'a besoin de spécifier la politique que quand un EmailMessage est créé, soit en instanciant directement un EmailMessage pour créer un nouvel e-mail, soit lors de l'analyse d'un flux entrant en utilisant un parser. Mais la politique peut être changée quand le message est sérialisé en utilisant un generator. Cela permet, par exemple, d'analyser un message e-mail générique du disque, puis de le sérialiser en utilisant une configuration SMTP standard quand on l'envoie vers un serveur d'e-mail.

Le paquet *email* fait son maximum pour cacher les détails des différentes RFCs de référence à l'application. Conceptuellement, l'application doit être capable de traiter l'e-mail comme une arborescence structurée de texte Unicode et

de pièces jointes binaires, sans avoir à se préoccuper de leur représentation sérialisée. Dans la pratique, cependant, il est souvent nécessaire d'être conscient d'au moins quelques règles relatives aux messages MIME et à leur structure, en particulier les noms et natures des "types de contenus" et comment ils identifient les documents à plusieurs parties. Pour la plupart, cette connaissance devrait seulement être nécessaire pour des applications plus complexes, et même là, il devrait être question des structures de haut niveau et non des détails sur la manière dont elles sont représentées. Comme les types de contenus MIME sont couramment utilisés dans les logiciels internet modernes (et non uniquement les e-mails), les développeurs sont généralement familiers de ce concept.

La section suivante décrit les fonctionnalités du paquet `email`. Nous commençons avec le modèle d'objet `message`, qui est la principale interface qu'une application utilise, et continuons avec les composants `parser` et `generator`. Ensuite, nous couvrons les contrôles `policy`, qui complètent le traitement des principaux composants de la bibliothèque.

Les trois prochaines sections couvrent les exceptions que le paquet peut rencontrer et les imperfections (non-respect des RFCs) que le module `parser` peut détecter. Ensuite nous couvrons les sous-composants `headerregistry` et `contentmanager`, qui fournissent des outils pour faire des manipulations plus détaillées des en-têtes et du contenu, respectivement. Les deux composants contiennent des fonctionnalités adaptées pour traiter et produire des messages qui sortent de l'ordinaire, et elles documentent aussi leurs API pour pouvoir les étendre, ce qui ne manquera pas d'intéresser les applications avancées.

Ci-dessous se trouve un ensemble d'exemples d'utilisations des éléments fondamentaux des API couvertes dans les sections précédentes.

Ce que nous venons d'aborder constitue l'API moderne (compatible Unicode) du paquet *email*. Les sections restantes, commençant par la classe `Message`, couvrent l'API héritée `compat32` qui traite beaucoup plus directement des détails sur la manière dont les e-mails sont représentés. L'API `compat32` ne cache *pas* les détails des RFCs à l'application, mais pour les applications qui requièrent d'opérer à ce niveau, elle peut être un outil pratique. Cette documentation est aussi pertinente pour les applications qui utilisent toujours l'API `compat32` pour des raisons de rétrocompatibilité.

Modifié dans la version 3.6 : Documents réorganisés et réécrits pour promouvoir la nouvelle API `EmailMessage`/`EmailPolicy`.

Contenus de la documentation du paquet `email` :

20.1.1 `email.message` : Representing an email message

Code source : Lib/email/message.py

Ajouté dans la version 3.6 : [1]

La classe centrale du paquet `email` est la classe `EmailMessage`, importée du module `email.message`. C'est la classe mère du modèle d'objet `email`. `EmailMessage` fournit la fonctionnalité de base pour définir et interroger les champs d'en-tête, pour accéder au corps des messages et pour créer ou modifier des messages structurés.

Un message électronique se compose d'en-têtes (*headers*) et d'une *charge utile* (*payload* en anglais, également appelée *contenu – content* en anglais). Les en-têtes sont des noms et valeurs de champ de style **RFC 5322** ou **RFC 6532**, où le nom et la valeur du champ sont séparés par deux points. Les deux-points ne font partie ni du nom du champ ni de la valeur du champ. La charge utile peut être un simple message texte, un objet binaire ou une séquence structurée de sous-messages chacun avec son propre ensemble d'en-têtes et sa propre charge utile. Ce dernier type de charge utile est indiqué par le message ayant un type MIME tel que `multipart/*` ou `message/rfc822`.

Le modèle conceptuel fourni par un objet `EmailMessage` est celui d'un dictionnaire ordonné d'en-têtes couplé à une *charge utile* qui représente le corps **RFC 5322** du message, qui peut être une liste de sous-objets `EmailMessage`. En plus des méthodes de dictionnaire normales pour accéder aux noms et valeurs d'en-têtes, il existe des méthodes pour accéder à des informations spécialisées à partir des en-têtes (par exemple le type de contenu MIME), pour agir sur la charge utile, pour générer une version sérialisée du message et pour parcourir récursivement l'arborescence d'objets.

The `EmailMessage` dictionary-like interface is indexed by the header names, which must be ASCII values. The values of the dictionary are strings with some extra methods. Headers are stored and returned in case-preserving

1. Ajouté à l'origine dans 3.4 en tant que *module provisoire*. Les documents pour la classe de message héritée ont été déplacés dans *email.message.Message : représentation d'un message électronique à l'aide de l'API compat32*.

form, but field names are matched case-insensitively. The keys are ordered, but unlike a real dict, there can be duplicates. Additional methods are provided for working with headers that have duplicate keys.

La *charge utile* est soit une chaîne ou un objet d'octets, dans le cas d'objets de message simples, soit une liste d'objets `EmailMessage` pour les documents de conteneur MIME tels que `multipart/*` et les objets messages `message/rfc822`.

class email.message.**EmailMessage**(*policy=default*)

Si *policy* est spécifiée, Python utilise les règles qu'elle spécifie pour mettre à jour et sérialiser la représentation du message. Si *policy* n'est pas définie, Python utilise la politique `default`, qui suit les règles des RFC de messagerie sauf pour les fins de ligne (au lieu de `\r\n` indiqués par la RFC, il utilise les fins de ligne standard Python `\n`). Pour plus d'informations, consultez la documentation `policy`.

as_string(*unixfrom=False*, *maxheaderlen=None*, *policy=None*)

Return the entire message flattened as a string. When optional *unixfrom* is true, the envelope header is included in the returned string. *unixfrom* defaults to `False`. For backward compatibility with the base `Message` class *maxheaderlen* is accepted, but defaults to `None`, which means that by default the line length is controlled by the `max_line_length` of the policy. The *policy* argument may be used to override the default policy obtained from the message instance. This can be used to control some of the formatting produced by the method, since the specified *policy* will be passed to the `Generator`.

L'aplatissement du message peut déclencher des changements dans `EmailMessage` si les valeurs par défaut doivent être renseignées pour terminer la transformation en chaîne (par exemple, les limites MIME peuvent être générées ou modifiées).

Notez que cette méthode est fournie à titre de commodité et n'est peut-être pas la méthode la plus utile pour sérialiser les messages dans votre application, en particulier si vous traitez plusieurs messages. Voir `email.generator.Generator` pour une API plus flexible pour la sérialisation des messages. Notez également que cette méthode est limitée à la production de messages sérialisés en « 7 bits propres » lorsque `utf8` est `False`, qui est la valeur par défaut.

Modifié dans la version 3.6 : le comportement par défaut lorsque *maxheaderlen* n'est pas spécifié est passé de la valeur par défaut à 0 à la valeur par défaut de *max_line_length* de la politique.

__str__()

Équivalent à `as_string(policy=self.policy.clone(utf8=True))`. Permet à `str(msg)` de produire une chaîne contenant le message sérialisé dans un format lisible.

Modifié dans la version 3.4 : la méthode a été modifiée pour utiliser `utf8=True`, produisant ainsi une représentation de message semblable à **RFC 6531**, au lieu d'être un alias direct pour `as_string()`.

as_bytes(*unixfrom=False*, *policy=None*)

Renvoie le message entier aplati en tant qu'objet bytes. Lorsque l'option *unixfrom* est vraie, l'en-tête de l'enveloppe est inclus dans la chaîne renvoyée. *unixfrom* par défaut est `False`. L'argument *policy* peut être utilisé pour remplacer la politique par défaut obtenue à partir de l'instance de message. Cela peut être utilisé pour contrôler une partie du formatage produit par la méthode, puisque la *policy* spécifiée sera transmise à `BytesGenerator`.

L'aplatissement du message peut déclencher des changements dans `EmailMessage` si les valeurs par défaut doivent être renseignées pour terminer la transformation en chaîne (par exemple, les limites MIME peuvent être générées ou modifiées).

Notez que cette méthode est fournie à titre de commodité et n'est peut-être pas la méthode la plus utile pour sérialiser les messages dans votre application, en particulier si vous traitez plusieurs messages. Voir `email.generator.BytesGenerator` pour une API plus flexible pour la sérialisation des messages.

__bytes__()

Equivalent to `as_bytes()`. Allows `bytes(msg)` to produce a bytes object containing the serialized message.

is_multipart()

Renvoie `True` si la charge utile du message est une liste d'objets `EmailMessage`, sinon renvoie `False`. Lorsque `is_multipart()` renvoie `False`, la charge utile doit être un objet chaîne (qui peut être une

charge utile binaire encodée CTE). Notez que *is_multipart()* renvoyant `True` ne signifie pas nécessairement que `msg.get_content_maintype() == 'multipart'` renvoie `True`. Par exemple, `is_multipart` renvoie `True` lorsque le *EmailMessage* est de type `message/rfc822`.

set_unixfrom(*unixfrom*)

Définit l'en-tête de l'enveloppe du message sur *unixfrom*, qui doit être une chaîne (voir *mboxMessage* pour une brève description de cet en-tête).

get_unixfrom()

Renvoie l'en-tête de l'enveloppe du message. La valeur par défaut est `None` si l'en-tête de l'enveloppe n'a jamais été défini.

Les méthodes suivantes implémentent l'interface de type correspondance pour accéder aux en-têtes du message. Notez qu'il existe des différences sémantiques entre ces méthodes et une interface de correspondance normale (c'est-à-dire un dictionnaire). Par exemple, dans un dictionnaire, il n'y a pas de clés en double, mais ici, il peut y avoir des en-têtes de message en double. De plus, dans les dictionnaires, il n'y a pas d'ordre garanti pour les clés renvoyées par *keys()*, mais dans un objet *EmailMessage*, les en-têtes sont toujours renvoyés dans l'ordre dans lequel ils sont apparus dans le message d'origine, ou dans lequel ils ont été ajoutés au message plus tard. Tout en-tête supprimé puis rajouté est toujours ajouté à la fin de la liste des en-têtes.

Ces différences sémantiques sont intentionnelles et privilégient la commodité dans les cas d'utilisation les plus courants.

Notez que dans tous les cas, tout en-tête d'enveloppe présent dans le message n'est pas inclus dans l'interface de correspondance.

__len__()

Renvoie le nombre total d'en-têtes, y compris les doublons.

__contains__(*name*)

Renvoie `True` si l'objet message a un champ nommé *name*. La correspondance est effectuée sans tenir compte de la casse et *name* n'inclut pas les deux-points de fin. Utilisé pour l'opérateur `in`. Par exemple :

```
if 'message-id' in myMessage:
   print('Message-ID:', myMessage['message-id'])
```

__getitem__(*name*)

Renvoie la valeur du champ d'en-tête nommé. *name* n'inclut pas le séparateur de champ deux-points. Si l'en-tête est manquant, `None` est renvoyée ; *KeyError* n'est jamais levée.

Notez que si le champ nommé apparaît plus d'une fois dans les en-têtes du message, il n'est pas défini la valeur de quel champ est renvoyée. Utilisez la méthode *get_all()* pour obtenir les valeurs de tous les en-têtes existants nommés *name*.

En utilisant les politiques standard (non-`compat32`), la valeur renvoyée est une instance d'une sous-classe de *email.headerregistry.BaseHeader*.

__setitem__(*name*, *val*)

Ajoute un en-tête au message avec le nom de champ *name* et la valeur *val*. Le champ est ajouté à la fin des en-têtes existants du message.

Notez que cela n'écrase *pas* ou ne supprime aucun en-tête existant portant le même nom. Si vous voulez vous assurer que le nouvel en-tête est le seul présent dans le message avec le nom de champ *name*, supprimez d'abord le champ, par exemple :

```
del msg['subject']
msg['subject'] = 'Python roolz!'
```

If the *policy* defines certain headers to be unique (as the standard policies do), this method may raise a *ValueError* when an attempt is made to assign a value to such a header when one already exists. This behavior is intentional for consistency's sake, but do not depend on it as we may choose to make such assignments do an automatic deletion of the existing header in the future.

__delitem__(*name*)

Supprime toutes les occurrences du champ portant le nom *name* des en-têtes du message. Aucune exception n'est levée si le champ nommé n'est pas présent dans les en-têtes.

keys()

Renvoie une liste de tous les noms de champs d'en-tête du message.

values()

Renvoie une liste de toutes les valeurs de champ du message.

items()

Renvoie une liste de couples contenant tous les en-têtes et valeurs de champ du message.

get(*name*, *failobj=None*)

Return the value of the named header field. This is identical to `__getitem__()` except that optional *failobj* is returned if the named header is missing (*failobj* defaults to `None`).

Voici quelques méthodes supplémentaires utiles liées aux en-têtes :

get_all(*name*, *failobj=None*)

Renvoie la liste de toutes les valeurs du champ nommé *name*. S'il n'y a pas d'en-têtes nommés de ce type dans le message, *failobj* est renvoyé (la valeur par défaut est `None`).

add_header(*_name*, *_value*, ***_params*)

Étend les en-têtes. Cette méthode est similaire à *__setitem__()* sauf que des paramètres d'en-tête supplémentaires peuvent être fournis en tant qu'arguments nommés. *_name* est le champ d'en-tête à ajouter et *_value* est la valeur *primaire* de l'en-tête.

Pour chaque élément du dictionnaire d'arguments nommés *_params*, la clé est prise comme nom de paramètre, avec des traits de soulignement convertis en tirets (puisque les tirets sont illégaux dans les identifiants Python). Normalement, le paramètre est ajouté en tant que `key="value"` sauf si la valeur est `None`, auquel cas seule la clé est ajoutée.

Si la valeur contient des caractères non-ASCII, le jeu de caractères et la langue peuvent être explicitement contrôlés en spécifiant la valeur sous la forme d'un triplet au format `(CHARSET, LANGUAGE, VALUE)`, où `CHARSET` est une chaîne nommant le jeu de caractères à utiliser pour encoder la valeur, `LANGUAGE` peut généralement être défini sur `None` ou sur la chaîne vide (voir **RFC 2231** pour d'autres possibilités) et `VALUE` est la chaîne contenant les valeurs des points de code non-ASCII. Si un triplet n'est pas passé et que la valeur contient des caractères non-ASCII, elle est automatiquement encodée au format **RFC 2231** en utilisant `utf-8` comme `CHARSET` et `None` comme `LANGUAGE`.

Voici un exemple :

```
msg.add_header('Content-Disposition', 'attachment', filename='bud.gif')
```

Cela ajoute un en-tête qui ressemble à :

```
Content-Disposition: attachment; filename="bud.gif"
```

Un exemple d'interface étendue avec des caractères non-ASCII :

```
msg.add_header('Content-Disposition', 'attachment',
               filename=('iso-8859-1', '', 'Fußballer.ppt'))
```

replace_header(*_name*, *_value*)

Remplace un en-tête. Remplace le premier en-tête trouvé dans le message qui correspond à *_name*, en conservant l'ordre des en-têtes et la casse du nom de champ de l'en-tête d'origine. Si aucun en-tête correspondant n'est trouvé, lève une *KeyError*.

get_content_type()

Renvoie le type de contenu du message, contraint en minuscules de la forme *maintype/subtype*. S'il n'y a pas d'en-tête *Content-Type* dans le message, renvoie la valeur renvoyée par *get_default_type()*. Si l'en-tête *Content-Type* n'est pas valide, renvoie `text/plain`.

(Selon la **RFC 2045**, les messages ont toujours un type par défaut, `get_content_type()` renvoie toujours une valeur. La **RFC 2045** définit le type par défaut d'un message comme étant `text/plain` à moins qu'il n'apparaisse dans un conteneur `multipart/digest`, auquel cas ce serait `message/rfc822`. Si l'en-tête `Content-Type` a une spécification de type invalide, la **RFC 2045** exige que le type par défaut soit `text/plain`.)

get_content_maintype()

Renvoie le type de contenu principal du message. C'est la partie `maintype` de la chaîne renvoyée par `get_content_type()`.

get_content_subtype()

Renvoie le type de sous-contenu du message. C'est la partie `subtype` de la chaîne renvoyée par `get_content_type()`.

get_default_type()

Renvoie le type de contenu par défaut. La plupart des messages ont un type de contenu `text/plain` par défaut, à l'exception des messages qui sont des sous-parties des conteneurs `multipart/digest`. Ces sous-parties ont un type de contenu par défaut de `message/rfc822`.

set_default_type(*ctype*)

Définit le type de contenu par défaut. *ctype* doit être `text/plain` ou `message/rfc822`, bien que cela ne soit pas imposé. Le type de contenu par défaut n'est pas stocké dans l'en-tête `Content-Type`, il n'affecte donc que la valeur de retour des méthodes `get_content_type` lorsqu'aucun en-tête `Content-Type` n'est présent dans le message .

set_param(*param*, *value*, *header='Content-Type'*, *requote=True*, *charset=None*, *language=''*, *replace=False*)

Définit un paramètre dans l'en-tête `Content-Type`. Si le paramètre existe déjà dans l'en-tête, remplace sa valeur par *value*. Lorsque *header* est `Content-Type` (la valeur par défaut) et que l'en-tête n'existe pas encore dans le message, l'ajoute, définit sa valeur sur `text/plain` et ajoute la nouvelle valeur du paramètre. *header* facultatif spécifie un en-tête alternatif à `Content-Type`.

Si la valeur contient des caractères non ASCII, le jeu de caractères et la langue peuvent être explicitement spécifiés à l'aide des paramètres facultatifs *charset* et *language*. L'option *language* spécifie la langue **RFC 2231**, par défaut la chaîne vide. *charset* et *language* doivent être des chaînes. La valeur par défaut est `utf8` pour *charset* et `None` pour *language*.

Si *replace* est `False` (valeur par défaut), l'en-tête est déplacé à la fin de la liste des en-têtes. Si *replace* est `True`, l'en-tête est mis à jour sur place.

L'utilisation du paramètre *requote* avec les objets `EmailMessage` est obsolète.

Note that existing parameter values of headers may be accessed through the `params` attribute of the header value (for example, `msg['Content-Type'].params['charset']`).

Modifié dans la version 3.4 : le paramètre nommé `replace` a été ajouté.

del_param(*param*, *header='content-type'*, *requote=True*)

Supprime complètement le paramètre donné de l'en-tête `Content-Type`. L'en-tête est réécrit en place sans le paramètre ou sa valeur. L'option *header* spécifie une alternative à `Content-Type`.

L'utilisation du paramètre *requote* avec les objets `EmailMessage` est obsolète.

get_filename(*failobj=None*)

Renvoie la valeur du paramètre `filename` de l'en-tête `Content-Disposition` du message. Si l'en-tête n'a pas de paramètre `filename`, cette méthode revient à rechercher le paramètre `name` dans l'en-tête `Content-Type`. Si aucun n'est trouvé, ou si l'en-tête est manquant, alors *failobj* est renvoyé. La chaîne renvoyée est toujours sans guillemets selon `email.utils.unquote()`.

get_boundary(*failobj=None*)

Renvoie la valeur du paramètre `boundary` de l'en-tête `Content-Type` du message, ou *failobj* si l'en-tête est manquant ou n'a pas de paramètre `boundary`. La chaîne renvoyée est toujours sans guillemets selon `email.utils.unquote()`.

set_boundary(*boundary*)

Définit le paramètre `boundary` de l'en-tête `Content-Type` sur *boundary*. `set_boundary()` entoure *boundary* de guillemets si nécessaire. Une `HeaderParseError` est levée si l'objet message n'a pas d'en-tête `Content-Type`.

Notez que l'utilisation de cette méthode est légèrement différente de la suppression de l'ancien en-tête `Content-Type` et de l'ajout d'un nouveau avec la nouvelle délimitation via `add_header()`, car `set_boundary()` préserve l'ordre des en-têtes `Content-Type` dans la liste des en-têtes.

get_content_charset(*failobj=None*)

Renvoie le paramètre `charset` de l'en-tête `Content-Type`, contraint en minuscules. S'il n'y a pas d'en-tête `Content-Type`, ou si cet en-tête n'a pas de paramètre `charset`, *failobj* est renvoyé.

get_charsets(*failobj=None*)

Renvoie la liste des noms des jeux de caractères dans le message. Si le message est un `multipart`, alors la liste contient un élément pour chaque sous-partie dans la charge utile, sinon c'est une liste de longueur 1.

Chaque élément de la liste est une chaîne qui est la valeur du paramètre `charset` dans l'en-tête `Content-Type` pour la sous-partie représentée. Si la sous-partie n'a pas d'en-tête `Content-Type`, pas de paramètre `charset` ou n'est pas du type MIME principal `text`, alors cet élément est *failobj* dans la liste renvoyée.

is_attachment()

Renvoie `True` s'il y a un en-tête `Content-Disposition` et que sa valeur (insensible à la casse) est `attachment`, `False` sinon.

Modifié dans la version 3.4.2 : *is_attachment* est maintenant une méthode au lieu d'une propriété, par souci de cohérence avec `is_multipart()`.

get_content_disposition()

Renvoie la valeur en minuscules (sans paramètres) de l'en-tête `Content-Disposition` du message s'il en a un, ou `None`. Les valeurs possibles pour cette méthode sont *inline*, *attachment* ou `None` si le message respecte la **RFC 2183**.

Ajouté dans la version 3.5.

Les méthodes suivantes concernent l'interrogation et la manipulation du contenu (charge utile) du message.

walk()

La méthode `walk()` est un générateur polyvalent qui peut être utilisé pour itérer sur toutes les parties et sous-parties d'un arbre d'objets de message, dans l'ordre de parcours en profondeur d'abord. L'utilisation classique est d'itérer avec `walk()` dans une boucle `for` ; chaque itération renvoie la sous-partie suivante.

Voici un exemple qui imprime le type MIME de chaque partie d'une structure de message en plusieurs parties :

```
>>> for part in msg.walk():
...     print(part.get_content_type())
multipart/report
text/plain
message/delivery-status
text/plain
text/plain
message/rfc822
text/plain
```

`walk` itère sur les sous-parties de toute partie où `is_multipart()` renvoie `True`, même si `msg.get_content_maintype() == 'multipart'` peut renvoyer `False`. Nous pouvons le voir dans notre exemple en utilisant la fonction d'aide au débogage `_structure` :

```
>>> from email.iterators import _structure
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> for part in msg.walk():
...     print(part.get_content_maintype() == 'multipart',
...           part.is_multipart())
True True
False False
False True
False False
False False
False True
False False
>>> _structure(msg)
multipart/report
    text/plain
    message/delivery-status
        text/plain
        text/plain
    message/rfc822
        text/plain
```

Ici, les parties `message` ne sont pas des `multiparts`, mais elles contiennent des sous-parties. `is_multipart()` renvoie `True` et `walk` descend dans les sous-parties.

get_body(*preferencelist=('related', 'html', 'plain')*)

Renvoie la partie MIME qui est la meilleure candidate pour être le corps du message.

preferencelist doit être une séquence de chaînes de l'ensemble `related`, `html` et `plain`, et indique l'ordre de préférence pour le type de contenu de la partie renvoyée.

Elle commence par rechercher des correspondances candidates avec l'objet sur lequel la méthode `get_body` est appelée.

Si `related` n'est pas inclus dans *preferencelist*, elle considère la partie racine (ou sous-partie de la partie racine) de tout lien rencontré comme candidat si la (sous-)partie correspond à une préférence.

Lorsqu'elle rencontre un `multipart/related`, elle vérifie le paramètre `start` et si une partie avec un *`Content-ID`* correspondant est trouvée, elle la considère uniquement lors de la recherche de correspondances candidates. Sinon, elle ne considère que la première partie (racine par défaut) de `multipart/related`.

Si une partie a un en-tête *`Content-Disposition`*, elle ne considère la partie comme une correspondance candidate que si la valeur de l'en-tête est `inline`.

Si aucun des candidats ne correspond à aucune des préférences dans *preferencelist*, elle renvoie `None`.

Remarques : (1) Pour la plupart des applications, les seules combinaisons *preferencelist* qui ont vraiment un sens sont `('plain',)`, `('html', 'plain')` et la valeur par défaut `( 'related', 'html', 'plain')`. (2) Parce que la correspondance commence avec l'objet sur lequel `get_body` est appelée, appeler `get_body` sur un `multipart/related` renvoie l'objet lui-même à moins que *preferencelist* n'ait une valeur autre que celle par défaut. (3) Les messages (ou parties de message) qui ne spécifient pas un *`Content-Type`* ou dont l'en-tête *`Content-Type`* est invalide sont traités comme s'ils étaient de type `text/plain`, ce qui peut occasionnellement amener `get_body` à renvoyer des résultats inattendus.

iter_attachments()

Renvoie un itérateur sur toutes les sous-parties immédiates du message qui ne sont pas des parties de « corps » candidates. Autrement dit, elle ignore la première occurrence de chacun des éléments suivants : `text/plain`, `text/html`, `multipart/related` ou `multipart/alternative` (sauf s'ils sont explicitement marqués comme pièces jointes via *`Content-Disposition: attachment`*), et renvoie toutes les parties restantes. Lorsqu'elle est appliquée directement à un `multipart/related`, renvoie un itérateur sur toutes les parties liées sauf la partie racine (c'est-à-dire la partie pointée par le paramètre `start`, ou la première partie s'il n'y a pas de paramètre `start` ou si le paramètre `start` ne correspond pas au *`Content-ID`* de l'une des parties). Lorsqu'elle est appliquée directement à un `multipart/`

`alternative` ou à un non-`multipart`, elle renvoie un itérateur vide.

iter_parts()

Renvoie un itérateur sur toutes les sous-parties immédiates du message, qui seront vides pour une non-`multipart` (voir aussi `walk()`).

get_content(**args*, *content_manager=None*, ***kw*)

Appelle la méthode `get_content()` du *content_manager*, en passant *self* comme objet message et en passant tout autre argument ou mot-clé comme argument supplémentaire. Si *content_manager* n'est pas spécifié, utilise le `content_manager` spécifié par la `policy` actuelle.

set_content(**args*, *content_manager=None*, ***kw*)

Appelle la méthode `set_content()` du *content_manager*, en passant *self* comme objet message et en passant tout autre argument ou mot-clé comme argument supplémentaire. Si *content_manager* n'est pas spécifié, utilise le `content_manager` spécifié par la `policy` actuelle.

make_related(*boundary=None*)

Convertit un message non-`multipart` en un message `multipart/related`, en déplaçant tous les en-têtes `Content-` existants et la charge utile dans une (nouvelle) première partie du `multipart`. Si *boundary* est spécifié, elle l'utilise comme chaîne de délimitation dans le *multipart*, sinon elle laisse la délimitation être créée automatiquement lorsque cela est nécessaire (par exemple, lorsque le message est sérialisé).

make_alternative(*boundary=None*)

Convertit un non-`multipart` ou un `multipart/related` en un `multipart/alternative`, en déplaçant tous les en-têtes `Content-` existants et la charge utile dans une (nouvelle) première partie du `multipart`. Si *boundary* est spécifiée, l'utilise comme chaîne de délimitation dans le *multipart*, sinon laisse la délimitation être créée automatiquement lorsque cela est nécessaire (par exemple, lorsque le message est sérialisé).

make_mixed(*boundary=None*)

Convertit un non-`multipart`, un `multipart/related` ou un `multipart-alternative` en un `multipart/mixed`, en déplaçant tous les en-têtes `Content-` existants et charge utile dans une (nouvelle) première partie du `multipart`. Si *boundary* est spécifiée, l'utilise comme chaîne de délimitation dans le *multipart*, sinon elle laisse la délimitation être créée automatiquement lorsque cela est nécessaire (par exemple, lorsque le message est sérialisé).

add_related(**args*, *content_manager=None*, ***kw*)

Si le message est un `multipart/related`, crée un nouvel objet message, passe tous les arguments à sa méthode `set_content()` et le joint avec `attach()` au `multipart`. Si le message n'est pas en `multipart`, appelle `make_related()` puis procède comme ci-dessus. Si le message est un autre type de `multipart`, lève une `TypeError`. Si *content_manager* n'est pas spécifié, utilise le `content_manager` spécifié par la `policy` actuelle. Si la partie ajoutée n'a pas d'en-tête `Content-Disposition`, en ajoute un avec la valeur `inline`.

add_alternative(**args*, *content_manager=None*, ***kw*)

Si le message est un `multipart/alternative`, crée un nouvel objet message, transmet tous les arguments à sa méthode `set_content()` et le joint avec `attach()` au `multipart`. Si le message est non-`multipart` ou `multipart/related`, appelle `make_alternative()` puis procède comme ci-dessus. Si le message est un autre type de `multipart`, lève une `TypeError`. Si *content_manager* n'est pas spécifié, utilise le `content_manager` spécifié par la `policy` actuelle.

add_attachment(**args*, *content_manager=None*, ***kw*)

Si le message est un `multipart/mixed`, crée un nouvel objet message, passe tous les arguments à sa méthode `set_content()` et le joint avec `attach()` au `multipart`. Si le message n'est pas en `multipart`, `multipart/related` ou `multipart/alternative`, appelle `make_mixed()` puis procède comme ci-dessus. Si *content_manager* n'est pas spécifié, utilise le `content_manager` spécifié par la `policy` actuelle. Si la partie ajoutée n'a pas d'en-tête `Content-Disposition`, en ajoute un avec la valeur `attachment`. Cette méthode peut être utilisée à la fois pour les pièces jointes explicites (`Content-Disposition:attachment`) et les pièces jointes `inline` (`Content-Disposition:inline`), en passant les options appropriées au `content_manager`.

clear()

Supprime la charge utile et tous les en-têtes.

clear_content()

Remove the payload and all of the *!Content-* headers, leaving all other headers intact and in their original order.

Les objets *EmailMessage* ont les attributs d'instance suivants :

preamble

Le format d'un document MIME permet d'insérer du texte entre la ligne vide suivant les en-têtes et la première chaîne de délimitation en plusieurs parties. Normalement, ce texte n'est jamais visible dans un lecteur de courrier compatible MIME car il ne fait pas partie de l'attirail MIME standard. Toutefois, lors de l'affichage du texte brut du message ou lors de l'affichage du message dans un lecteur non compatible MIME, ce texte peut devenir visible.

L'attribut *preamble* contient ce texte hors cadre de tête pour les documents MIME. Lorsque *Parser* découvre du texte après les en-têtes mais avant la première chaîne de délimitation, il attribue ce texte à l'attribut *preamble* du message. Lorsque *Generator* écrit la représentation en texte brut d'un message MIME, et qu'il trouve que le message a un attribut *preamble*, il écrit ce texte dans la zone entre les en-têtes et la première limite. Voir *email.parser* et *email.generator* pour plus de détails.

Notez que si l'objet message n'a pas de préambule, l'attribut *preamble* est `None`.

epilogue

L'attribut *epilogue* agit de la même manière que l'attribut *preamble*, sauf qu'il contient du texte qui apparaît entre la dernière limite et la fin du message. Comme avec *preamble*, s'il n'y a pas de texte d'épilogue, cet attribut est `None`.

defects

L'attribut *defects* contient une liste de tous les problèmes rencontrés lors de l'analyse de ce message. Voir *email.errors* pour une description détaillée des défauts d'analyse possibles.

class email.message.**MIMEPart**(*policy=default*)

Cette classe représente une sous-partie d'un message MIME. Elle est identique à *EmailMessage*, sauf qu'aucun en-tête *MIME-Version* n'est ajouté lorsque *set_content()* est appelée, car les sous-parties n'ont pas besoin de leurs propres en-têtes *MIME-Version*.

Notes

20.1.2 email.parser : Parsing email messages

Code source : Lib/email/parser.py

Les instances de messages peuvent être créées de deux façons : elles peuvent être créées de toutes pièces en créant un objet `EmailMessage`, en ajoutant des en-têtes en utilisant l'interface de dictionnaire, et en ajoutant un ou plusieurs corps de message en utilisant `set_content()` et les méthodes associées, ou elles peuvent être créées en analysant une représentation sérialisée de l'e-mail.

Le paquet *email* fournit un analyseur standard qui comprend la plupart des structures de documents de courrier électronique, y compris les documents MIME. Vous pouvez passer à l'analyseur un objet *bytes*, chaîne ou fichier, et l'analyseur vous renverra l'instance racine *EmailMessage* de la structure de l'objet. Pour les messages simples non MIME, la charge utile de cet objet racine sera probablement une chaîne contenant le texte du message. Pour les messages MIME, la méthode *is_multipart()* de l'objet racine renvoie `True`, et les sous-parties sont accessibles via les méthodes de manipulation de la charge utile, telles que `get_body()`, *iter_parts()* et *walk()*.

Il existe en fait deux interfaces d'analyseur disponibles, l'API *Parser* et l'API incrémentale *FeedParser*. L'API *Parser* est plus utile si vous avez le texte entier du message en mémoire ou si le message entier réside dans un fichier sur le système de fichiers. *FeedParser* est plus appropriée lorsque vous lisez le message à partir d'un flux qui peut bloquer en attente d'une entrée supplémentaire (comme la lecture d'un message électronique à partir d'un

connecteur réseau). `FeedParser` peut consommer et analyser le message de manière incrémentielle et ne renvoie l'objet racine que lorsque vous fermez l'analyseur.

Note that the parser can be extended in limited ways, and of course you can implement your own parser completely from scratch. All of the logic that connects the `email` package's bundled parser and the `EmailMessage` class is embodied in the `Policy` class, so a custom parser can create message object trees any way it finds necessary by implementing custom versions of the appropriate `Policy` methods.

API *FeedParser*

`BytesFeedParser`, importée du module `email.feedparser`, fournit une API propice à l'analyse incrémentielle des messages électroniques, adaptée à la lecture du texte d'un message électronique à partir d'une source qui peut bloquer (comme un connecteur). `BytesFeedParser` peut bien sûr être utilisée pour analyser un message électronique entièrement contenu dans un *objet octets-compatible*, une chaîne ou un fichier, mais l'API `BytesParser` peut être plus pratique dans ces utilisations. La sémantique et les résultats des deux API d'analyseurs sont identiques.

L'API de `BytesFeedParser` est simple ; vous créez une instance, l'alimentez d'une suite d'octets jusqu'à ce qu'il n'y en ait plus pour l'alimenter, puis fermez l'analyseur pour récupérer l'objet message racine. `BytesFeedParser` est extrêmement précise lors de l'analyse des messages conformes aux normes, et elle fait un très bon travail d'analyse des messages non conformes, fournissant des informations sur ce qu'elle considère comme non approprié dans un message. Elle remplit l'attribut `defects` d'un objet message avec une liste de tous les problèmes trouvés dans un message. Voir le module `email.errors` pour la liste des défauts qu'elle peut trouver.

Voici l'API de `BytesFeedParser` :

class email.parser.**BytesFeedParser**(*_factory=None*, *, *policy=policy.compat32*)

Crée une instance `BytesFeedParser`. *_factory* est un appelable facultatif sans argument ; s'il n'est pas spécifié, elle utilise la `message_factory` de la *policy*. Elle appelle *_factory* chaque fois qu'un nouvel objet de message est nécessaire.

Si *policy* est spécifiée, elle utilise les règles spécifiées pour mettre à jour la représentation du message. Si *policy* n'est pas définie, elle utilise la stratégie `compat32`, qui maintient la rétrocompatibilité avec la version Python 3.2 du paquet de messagerie et fournit `Message` comme usine par défaut. Toutes les autres stratégies fournissent `EmailMessage` comme *_factory* par défaut. Pour plus d'informations sur ce que *policy* régit, consultez la documentation `policy`.

Remarque : **Le paramètre nommé « policy » doit toujours être spécifié** ; La valeur par défaut deviendra `email.policy.default` dans une future version de Python.

Ajouté dans la version 3.2.

Modifié dans la version 3.3 : ajout du paramètre nommé *policy*.

Modifié dans la version 3.6 : *_factory* utilise par défaut la politique `message_factory`.

feed(*data*)

Alimente l'analyseur avec plus de données. *data* doit être un *objet octets-compatible* contenant une ou plusieurs lignes. Les lignes peuvent être partielles et l'analyseur assemble correctement ces lignes partielles. Les lignes peuvent avoir l'une des trois fins de ligne courantes : retour chariot, nouvelle ligne ou retour chariot et nouvelle ligne (elles peuvent même être mélangées).

close()

Termine l'analyse de toutes les données précédemment alimentées et renvoie l'objet message racine. Ce qui se passe si `feed()` est appelée après l'appel de cette méthode n'est pas défini.

class email.parser.**FeedParser**(*_factory=None*, *, *policy=policy.compat32*)

Fonctionne comme `BytesFeedParser` sauf que l'entrée de la méthode `feed()` doit être une chaîne. Ceci est d'une utilité limitée, car la seule façon pour qu'un tel message soit valide est qu'il ne contienne que du texte ASCII ou, si `utf8` vaut `True`, aucun binaire en pièce jointe.

Modifié dans la version 3.3 : ajout du paramètre nommé *policy*.

API de *Parser*

La classe `BytesParser`, importée du module `email.parser`, fournit une API qui peut être utilisée pour analyser un message lorsque le contenu complet du message est disponible dans un *objet octets-compatible* ou un fichier. Le module `email.parser` fournit également `Parser` pour l'analyse des chaînes et des analyseurs d'en-tête uniquement, `BytesHeaderParser` et `HeaderParser`, qui peuvent être utilisés si vous ne vous intéressez qu'aux en-têtes du message. `BytesHeaderParser` et `HeaderParser` peuvent être beaucoup plus rapides dans ces situations, car ils n'essaient pas d'analyser le corps du message.

class email.parser.**BytesParser**(*_class=None*, *, *policy=policy.compat32*)

Crée une instance `BytesParser`. Les arguments *_class* et *policy* ont la même signification et la même sémantique que les arguments *_factory* et *policy* de `BytesFeedParser`.

Remarque : **Le paramètre nommé « policy » doit toujours être spécifié** ; La valeur par défaut deviendra `email.policy.default` dans une future version de Python.

Modifié dans la version 3.3 : suppression de l'argument *strict* qui était obsolète dans 2.4. Ajout du paramètre nommé *policy*.

Modifié dans la version 3.6 : *_class* utilise par défaut la politique `message_factory`.

parse(*fp*, *headersonly=False*)

Lit toutes les données de l'objet de type fichier binaire *fp*, analyse les octets résultants et renvoie l'objet message. *fp* doit prendre en charge les méthodes `readline()` et `read()`.

Les octets contenus dans *fp* doivent être formatés comme un bloc d'en-têtes de style conforme à la **RFC 5322** (ou, si `utf8` est `True`, à la **RFC 6532**), éventuellement précédés d'un en-tête d'enveloppe. Le bloc d'en-têtes se termine soit par la fin des données, soit par une ligne blanche. Après le bloc d'en-têtes se trouve le corps du message (qui peut contenir des sous-parties codées MIME, y compris des sous-parties avec un `Content-Transfer-Encoding` de `8bit`).

headersonly est un indicateur facultatif spécifiant s'il faut arrêter l'analyse après avoir lu les en-têtes ou non. La valeur par défaut est `False`, ce qui signifie qu'il analyse tout le contenu du fichier.

parsebytes(*bytes*, *headersonly=False*)

Semblable à la méthode `parse()`, sauf qu'elle prend un *objet octets-compatible* au lieu d'un objet de type fichier. Appeler cette méthode sur un *objet octets-compatible* équivaut à envelopper *bytes* dans une instance `BytesIO` d'abord et à appeler `parse()`.

L'utilisation de l'option *headersonly* est identique à son utilisation dans la méthode `parse()`.

Ajouté dans la version 3.2.

class email.parser.**BytesHeaderParser**(*_class=None*, *, *policy=policy.compat32*)

Exactement comme `BytesParser`, sauf que *headersonly* par défaut est `True`.

Ajouté dans la version 3.3.

class email.parser.**Parser**(*_class=None*, *, *policy=policy.compat32*)

Cette classe est semblable à `BytesParser`, mais elle accepte une chaîne en entrée.

Modifié dans la version 3.3 : suppression de l'argument *strict*. Ajout de l'argument nommé *policy*.

Modifié dans la version 3.6 : *_class* utilise par défaut la politique `message_factory`.

parse(*fp*, *headersonly=False*)

Lit toutes les données de l'objet de type fichier en mode texte *fp*, analyse le texte résultant et renvoie l'objet message racine. *fp* doit prendre en charge les méthodes `readline()` et `read()` sur les objets de type fichier.

Outre l'exigence du mode texte, cette méthode fonctionne comme `BytesParser.parse()`.

parsestr(*text*, *headersonly=False*)

Similaire à la méthode `parse()`, sauf qu'elle prend un objet chaîne au lieu d'un objet de type fichier. Appeler cette méthode sur une chaîne équivaut à envelopper *text* dans une instance `StringIO` d'abord et à appeler `parse()`.

L'utilisation de l'option *headersonly* est identique à son utilisation dans la méthode `parse()`.

class email.parser.**HeaderParser**(*_class=None*, *, *policy=policy.compat32*)

Exactement comme `Parser`, sauf que *headersonly* par défaut est `True`.

Étant donné que la création d'une structure d'objet message à partir d'une chaîne ou d'un objet fichier est une tâche très courante, quatre fonctions sont fournies par commodité. Elles sont disponibles dans l'espace de noms de paquet de niveau supérieur `email`.

email.**message_from_bytes**(*s*, *_class=None*, *, *policy=policy.compat32*)

Renvoie une structure d'objet message à partir d'un *objet octets-compatible*. C'est équivalent à `BytesParser().parsebytes(s)`. *_class* et *policy* (facultatifs) sont interprétés comme pour le constructeur de classe `BytesParser`.

Ajouté dans la version 3.2.

Modifié dans la version 3.3 : suppression de l'argument *strict*. Ajout de l'argument nommé *policy*.

email.**message_from_binary_file**(*fp*, *_class=None*, *, *policy=policy.compat32*)

Renvoie une arborescence d'objets message à partir d'un *objet fichier* binaire ouvert. C'est équivalent à `BytesParser().parse(fp)`. *_class* et *policy* sont interprétés comme pour le constructeur de classe `BytesParser`.

Ajouté dans la version 3.2.

Modifié dans la version 3.3 : suppression de l'argument *strict*. Ajout de l'argument nommé *policy*.

email.**message_from_string**(*s*, *_class=None*, *, *policy=policy.compat32*)

Renvoie une structure d'objet message à partir d'une chaîne. C'est équivalent à `Parser().parsestr(s)`. *_class* et *policy* sont interprétés comme avec le constructeur de classe `Parser`.

Modifié dans la version 3.3 : suppression de l'argument *strict*. Ajout de l'argument nommé *policy*.

email.**message_from_file**(*fp*, *_class=None*, *, *policy=policy.compat32*)

Renvoie une arborescence de structures d'objets messages à partir d'un *objet fichier* ouvert. C'est équivalent à `Parser().parse(fp)`. *_class* et *policy* sont interprétés comme avec le constructeur de classe `Parser`.

Modifié dans la version 3.3 : suppression de l'argument *strict*. Ajout de l'argument nommé *policy*.

Modifié dans la version 3.6 : *_class* utilise par défaut la politique `message_factory`.

Voici un exemple d'utilisation `message_from_bytes()` dans une invite Python interactive :

```
>>> import email
>>> msg = email.message_from_bytes(myBytes)
```

Notes complémentaires

Voici des remarques sur la sémantique d'analyse :

- La plupart des messages de type non-`multipart` sont analysés comme un seul objet de message avec une charge utile de type chaîne. Ces objets renvoient `False` pour `is_multipart()` et `iter_parts()` donne une liste vide.
- Tous les messages de type `multipart` sont analysés comme un objet de message conteneur avec une liste d'objets de sous-messages pour leur charge utile. Le message du conteneur externe renvoie `True` pour `is_multipart()` et `iter_parts()` donne une liste de sous-parties.
- La plupart des messages avec un type de contenu de `message/*` (tels que `message/delivery-status` et `message/rfc822`) sont également analysés en tant qu'objet conteneur contenant une charge utile de type « liste de longueur 1 ». Leur méthode `is_multipart()` renvoie `True`. L'élément unique généré par `iter_parts()` est un objet sous-message.
- Certains messages non conformes aux normes peuvent ne pas être cohérents en interne quant à `multipart`. De tels messages peuvent avoir un en-tête `Content-Type` de type `multipart`, mais leur méthode `is_multipart()` peut renvoyer `False`. Si de tels messages ont été analysés avec la `FeedParser`, ils auront une instance de la classe `MultipartInvariantViolationDefect` dans leur liste d'attributs *defects*. Voir `email.errors` pour plus de détails.

20.1.3 email.generator : Generating MIME documents

Code source : Lib/email/igenerator.py

One of the most common tasks is to generate the flat (serialized) version of the email message represented by a message object structure. You will need to do this if you want to send your message via `smtplib.SMTP.sendmail()`, or print the message on the console. Taking a message object structure and producing a serialized representation is the job of the generator classes.

Comme avec le module `email.parser`, vous n'êtes pas limité aux fonctionnalités du générateur fourni ; vous pourriez en écrire un à partir de zéro vous-même. Cependant, le générateur fourni sait comment générer la plupart des e-mails d'une manière conforme aux normes, est conçu pour gérer correctement les e-mails MIME et non MIME, et pour que les opérations (sur des suites d'octets) d'analyse et de génération soient inverses, en supposant que la même `policy` est utilisée pour les deux. Autrement dit, l'analyse du flux d'octets sérialisé via la classe `BytesParser` puis la régénération du flux d'octets sérialisé à l'aide de `BytesGenerator` devrait produire une sortie identique à l'entrée [1] (d'un autre côté, l'utilisation du générateur sur un `EmailMessage` construit par programme peut entraîner des modifications de l'objet `EmailMessage` car les valeurs par défaut sont renseignées).

La classe `Generator` peut être utilisée pour aplatir un message dans une représentation sérialisée textuelle (par opposition à binaire), mais comme Unicode ne peut pas représenter directement des données binaires, le message est nécessairement transformé en quelque chose qui ne contient que des caractères ASCII, en utilisant les techniques standard de codage de transfert de contenu RFC pour le codage des messages électroniques pour le transport sur des canaux qui ne sont pas « complétement 8 bits ».

Pour s'adapter au traitement reproductible des messages signés SMIME `Generator` ne raccourcit pas les en-têtes pour les parties de message de type `multipart/signed` et toutes les sous-parties.

class email.generator.**BytesGenerator**(*outfp*, *mangle_from_=None*, *maxheaderlen=None*, *, *policy=None*)

Renvoie un objet `BytesGenerator` qui écrit tout message fourni à la méthode `flatten()`, ou tout texte encodé par *surrogateescape* fourni à la méthode `write()`, dans le *objet simili-fichier outfp*. *outfp* doit disposer d'une méthode `write` qui accepte les données binaires.

Si l'option *mangle_from_* est `True`, place un caractère > devant toute ligne du corps commençant par la chaîne exacte `"From "`, c'est-à-dire une ligne commençant par `From` suivi par une espace. *mangle_from_* prend par défaut la valeur du paramètre `mangle_from_` de la *policy* (qui est `True` pour la politique `compat32` et `False` pour tous les autres). *mangle_from_* est destiné à être utilisé lorsque les messages sont stockés au format Unix *mbox* (voir `mailbox` et WHY THE CONTENT-LENGTH FORMAT IS BAD).

Si *maxheaderlen* n'est pas `None`, raccourcit toutes les lignes d'en-tête qui sont plus longues que *maxheaderlen*, ou si `0`, ne raccourcit aucun en-tête. Si *manheaderlen* est `None` (valeur par défaut), formate les en-têtes et autres lignes de message en fonction des paramètres de *policy*.

Si *policy* est spécifiée, utilise cette politique pour contrôler la génération des messages. Si *policy* est `None` (par défaut), utilise la politique associée à l'objet `Message` ou `EmailMessage` passée à `flatten` pour contrôler la génération des messages. Voir `email.policy` pour plus de détails sur ce que *policy* contrôle.

Ajouté dans la version 3.2.

Modifié dans la version 3.3 : ajout du mot-clé *policy*.

Modifié dans la version 3.6 : le comportement par défaut des paramètres *mangle_from_* et *maxheaderlen* est de suivre la politique.

flatten(*msg*, *unixfrom=False*, *linesep=None*)

Affiche la représentation textuelle de la structure de l'objet message dont la racine est *msg* dans le fichier de sortie spécifié lors de la création de l'instance `BytesGenerator`.

Si l'option `cte_type` de `policy` est `8bit` (valeur par défaut), copie tous les en-têtes du message analysé d'origine qui n'ont pas été modifiés en sortie avec tous les octets dont le bit de poids fort identique

1. This statement assumes that you use the appropriate setting for `unixfrom`, and that there are no `email.policy` settings calling for automatic adjustments (for example, `refold_source` must be `none`, which is *not* the default). It is also not 100% true, since if the message does not conform to the RFC standards occasionally information about the exact original text is lost during parsing error recovery. It is a goal to fix these latter edge cases when possible.

à l'original, et conserve le *Content-Transfer-Encoding* non-ASCII de toutes les parties du corps qui en ont. Si `cte_type` est `7bit`, convertit les octets avec le bit de poids fort défini selon les besoins en utilisant un *Content-Transfer-Encoding* compatible ASCII. Autrement dit, transforme les parties non ASCII *Content-Transfer-Encoding* (*Content-Transfer-Encoding: 8bit*) en un *Content-Transfer-Encoding* compatible ASCII, et encode les octets non ASCII non valides de la RFC dans les en-têtes utilisant le jeu de caractères MIME `unknown-8bit`, les rendant ainsi conformes à la RFC.

Si *unixfrom* est `True`, affiche le délimiteur d'en-tête d'enveloppe utilisé par le format de boîte aux lettres Unix (voir *mailbox*) avant le premier des en-têtes **RFC 5322** de l'objet message racine. Si l'objet racine n'a pas d'en-tête d'enveloppe, en crée un standard. La valeur par défaut est `False`. Notez que pour les sous-parties, aucun en-tête d'enveloppe n'est jamais imprimé.

Si *linesep* n'est pas `None`, l'utilise comme caractère de séparation entre toutes les lignes du message aplati. Si *linesep* est `None` (valeur par défaut), utilise la valeur spécifiée dans la *policy*.

clone(*fp*)

Renvoie un clone indépendant de cette instance *BytesGenerator* avec exactement les mêmes paramètres d'option, et *fp* comme nouveau *outfp*.

write(*s*)

Encode *s* en utilisant le codec `ASCII` et le gestionnaire d'erreurs `surrogateescape`, et le passe à la méthode *write* du *outfp* passé au constructeur de *BytesGenerator*.

Par commodité, *EmailMessage* fournit les méthodes *as_bytes()* et `bytes(aMessage)` (alias `__bytes__()`), qui simplifient la génération d'une représentation binaire sérialisée d'un objet message. Pour plus de détails, voir *email.message*.

Comme les chaînes ne peuvent pas représenter des données binaires, la classe *Generator* doit convertir toutes les données binaires de tout message qu'elle aplatit en un format compatible ASCII, en les convertissant en un *Content-Transfer_Encoding* compatible ASCII. En utilisant la terminologie des RFC des e-mails, vous pouvez considérer cela comme un *Generator* sérialisant vers un flux d'entrées-sorties qui n'est pas « complètement 8 bits ». En d'autres termes, la plupart des applications voudront utiliser *BytesGenerator* et pas *Generator*.

class email.generator.**Generator**(*outfp*, *mangle_from_=None*, *maxheaderlen=None*, *, *policy=None*)

Renvoie un objet *Generator* qui écrit tout message fourni à la méthode *flatten()*, ou tout texte fourni à la méthode *write()*, dans l'*objet simili-fichier outfp*. *outfp* doit prendre en charge une méthode `write` qui accepte les données de type chaîne.

Si l'option *mangle_from_* est `True`, place un caractère > devant toute ligne du corps commençant par la chaîne exacte `"From "`, c'est-à-dire une ligne commençant par `From` suivi par une espace. *mangle_from_* prend par défaut la valeur du paramètre *mangle_from_* de la *policy* (qui est `True` pour la politique *compat32* et `False` pour tous les autres). *mangle_from_* est destiné à être utilisé lorsque les messages sont stockés au format Unix *mbox* (voir *mailbox* et WHY THE CONTENT-LENGTH FORMAT IS BAD).

Si *maxheaderlen* n'est pas `None`, raccourcit toutes les lignes d'en-tête qui sont plus longues que *maxheaderlen*, ou si `0`, ne raccourcit aucun en-tête. Si *manheaderlen* est `None` (valeur par défaut), formate les en-têtes et autres lignes de message en fonction des paramètres de *policy*.

Si *policy* est spécifiée, utilise cette politique pour contrôler la génération des messages. Si *policy* est `None` (par défaut), utilise la politique associée à l'objet *Message* ou *EmailMessage* passée à `flatten` pour contrôler la génération des messages. Voir *email.policy* pour plus de détails sur ce que *policy* contrôle.

Modifié dans la version 3.3 : ajout du mot-clé *policy*.

Modifié dans la version 3.6 : le comportement par défaut des paramètres *mangle_from_* et *maxheaderlen* est de suivre la politique.

flatten(*msg*, *unixfrom=False*, *linesep=None*)

Affiche la représentation textuelle de la structure de l'objet message dont la racine est *msg* dans le fichier de sortie spécifié lors de la création de l'instance *Generator*.

Si l'option *policy cte_type* est `8bit`, génère le message comme si l'option était définie sur `7bit` (c'est nécessaire car les chaînes ne peuvent pas représenter des octets non ASCII). Convertit tous les

octets avec le bit de poids fort défini selon les besoins à l'aide d'un `Content-Transfer-Encoding` compatible ASCII. Autrement dit, transforme les parties non ASCII `Content-Transfer-Encoding` (`Content-Transfer-Encoding: 8bit`) en un `Content-Transfer-Encoding` compatible ASCII, et encode les octets non ASCII non valides RFC dans les en-têtes utilisant le jeu de caractères MIME `unknown-8bit`, les rendant ainsi conformes à la RFC.

Si *unixfrom* est `True`, affiche le délimiteur d'en-tête d'enveloppe utilisé par le format de boîte aux lettres Unix (voir `mailbox`) avant le premier des en-têtes **RFC 5322** de l'objet message racine. Si l'objet racine n'a pas d'en-tête d'enveloppe, en crée un standard. La valeur par défaut est `False`. Notez que pour les sous-parties, aucun en-tête d'enveloppe n'est jamais imprimé.

Si *linesep* n'est pas `None`, l'utilise comme caractère de séparation entre toutes les lignes du message aplati. Si *linesep* est `None` (valeur par défaut), utilise la valeur spécifiée dans la *policy*.

Modifié dans la version 3.2 : ajout de la prise en charge du ré-encodage des corps de message `8bit` et de l'argument *linesep*.

clone(*fp*)

Renvoie un clone indépendant de cette instance `Generator` avec exactement les mêmes options, et avec *fp* comme nouveau *outfp*.

write(*s*)

Écrit *s* dans la méthode *write* de *outfp* passé au constructeur de `Generator`. Cela fournit une API ressemblant suffisamment au type fichier pour les instances de `Generator` utilisées dans la fonction `print()`.

Par commodité, `EmailMessage` fournit les méthodes `as_string()` et `str(aMessage)` (alias `__str__()`), qui simplifient la génération d'une représentation sous forme de chaîne formatée d'un objet message. Pour plus de détails, voir `email.message`.

Le module `email.generator` fournit également une classe dérivée, `DecodedGenerator`, qui ressemble à la classe de base `Generator`, sauf que les parties non-`text` ne sont pas sérialisées, mais sont plutôt représentées dans le flux de sortie par une chaîne dérivée d'un modèle rempli d'informations sur la partie concernée.

class email.generator.**DecodedGenerator**(*outfp*, *mangle_from_=None*, *maxheaderlen=None*, *fmt=None*, **, policy=None*)

Agit comme `Generator`, sauf que pour toute sous-partie du message transmise à `Generator.flatten()`, si la sous-partie est de type principal `text`, affiche la charge utile décodée de la sous-partie, et si le type principal n'est pas `text`, au lieu de l'afficher, remplit la chaîne *fmt* en utilisant les informations de la partie et affiche la chaîne remplie résultante.

Pour remplir *fmt*, exécute `fmt % part_info`, où `part_info` est un dictionnaire composé des clés et valeurs suivantes :
— `type` – Type MIME complet de la partie non-`text`
— `maintype` – Type MIME principal de la partie non-`text`
— `subtype` – Sous-type MIME de la partie non-`text`
— `filename` – Nom de fichier de la partie non-`text`
— `description` – Description associée à la partie non-`text`
— `encoding` – Encodage du contenu de la partie non-`text`
Si *fmt* est `None`, utilise le *fmt* par défaut suivant :

"[Non-text (%(type)s) part of message omitted, filename %(filename)s]"

*_mangle_from_* et *maxheaderlen* sont optionnels et fonctionnent comme avec la classe mère `Generator`.

Notes

20.1.4 email.policy : Policy Objects

Ajouté dans la version 3.3.

Code source : Lib/email/policy.py

Le principal objectif du paquet `email` est la gestion des messages électroniques comme décrit par les diverses RFC de messagerie et MIME. Cependant, le format général des messages électroniques (un bloc de champs d'en-tête

composés chacun d'un nom suivi de deux-points suivi d'une valeur, le bloc entier suivi d'une ligne vide et d'un « corps » arbitraire), est un format qui a trouvé son utilité en dehors du domaine du courrier électronique. Certaines de ces utilisations sont assez conformes aux principales RFC de messagerie, d'autres non. Même lorsque vous travaillez avec des e-mails, il est parfois souhaitable de ne pas respecter strictement les RFC, par exemple en générant des e-mails qui interagissent avec des serveurs de messagerie qui ne respectent pas eux-mêmes les normes ou qui implémentent des extensions que vous souhaitez utiliser d'une manière qui enfreint les normes.

Les objets de définition de politique donnent au paquet de messagerie la flexibilité nécessaire pour gérer tous ces cas d'utilisation disparates.

Un objet `Policy` encapsule un ensemble d'attributs et de méthodes qui contrôlent le comportement de divers composants du paquet de messagerie lors de son utilisation. Les instances `Policy` peuvent être transmises à diverses classes et méthodes dans le paquet de courrier électronique pour modifier le comportement par défaut. Les valeurs réglables et leurs valeurs par défaut sont décrites ci-dessous.

Il existe une politique par défaut utilisée par toutes les classes dans le paquet de messagerie. Pour toutes les classes `parser` et les fonctions pratiques associées, et pour la classe `Message`, il s'agit de la politique `Compat32`, via son instance correspondante prédéfinie `compat32`. Cette politique fournit une rétrocompatibilité complète (dans certains cas, y compris la compatibilité des bogues) avec la version pré-Python3.3 du paquet de messagerie.

Cette valeur par défaut pour le paramètre nommé *policy* de `EmailMessage` est la politique `EmailPolicy`, via son instance prédéfinie `default`.

Lorsqu'un objet `Message` ou `EmailMessage` est créé, il acquiert une politique. Si le message est créé par un `parser`, la politique transmise à l'analyseur est la politique utilisée par le message qu'il crée. Si le message est créé par le programme, la stratégie peut être spécifiée lors de sa création. Lorsqu'un message est passé à un `generator`, le générateur utilise la politique du message par défaut, mais vous pouvez également passer une politique spécifique au générateur qui remplace celle stockée sur l'objet message.

La valeur par défaut du paramètre nommé *policy* pour les classes `email.parser` et les fonctions pratiques de l'analyseur **va changer** dans une future version de Python. Par conséquent, vous devez **toujours spécifier explicitement la stratégie que vous souhaitez utiliser** lors de l'appel de l'une des classes et fonctions décrites dans le module `parser`.

La première partie de cette documentation couvre les fonctionnalités de `Policy`, une *classe mère abstraite* qui définit les fonctionnalités communes à tous les objets de stratégie, y compris `compat32`. Cela inclut certaines méthodes automatiques (*hooks* en anglais) appelées en interne par le paquet de messagerie, qu'une stratégie personnalisée peut remplacer pour obtenir un comportement différent. La deuxième partie décrit les classes concrètes `EmailPolicy` et `Compat32`, qui implémentent les points d'entrée automatiques qui fournissent respectivement le comportement standard et le comportement et les fonctionnalités rétrocompatibles.

Les instances `Policy` sont immuables, mais elles peuvent être clonées, en acceptant les mêmes arguments nommés que le constructeur de classe et en renvoyant une nouvelle instance `Policy` qui est une copie de l'original mais avec les valeurs d'attributs spécifiées modifiées.

Par exemple, le code suivant peut être utilisé pour lire un message électronique à partir d'un fichier sur le disque et le transmettre au programme système `sendmail` sur un système Unix :

```
>>> from email import message_from_binary_file
>>> from email.generator import BytesGenerator
>>> from email import policy
>>> from subprocess import Popen, PIPE
>>> with open('mymsg.txt', 'rb') as f:
...     msg = message_from_binary_file(f, policy=policy.default)
...
>>> p = Popen(['sendmail', msg['To'].addresses[0]], stdin=PIPE)
>>> g = BytesGenerator(p.stdin, policy=msg.policy.clone(linesep='\r\n'))
>>> g.flatten(msg)
>>> p.stdin.close()
>>> rc = p.wait()
```

Ici, nous disons à `BytesGenerator` d'utiliser les caractères de séparation de ligne corrects de la RFC lors de la création de la chaîne binaire à fournir à `stdin` de `sendmail`, où la politique par défaut utiliserait `\n` comme séparateur

de ligne.

Certaines méthodes du paquet de messagerie acceptent un argument nommé *policy*, permettant à la politique d'être remplacée pour cette méthode. Par exemple, le code suivant utilise la méthode `as_bytes()` de l'objet *msg* de l'exemple précédent et écrit le message dans un fichier en utilisant les séparateurs de ligne natifs de la plate-forme sur laquelle il est en cours d'exécution :

```
>>> import os
>>> with open('converted.txt', 'wb') as f:
...     f.write(msg.as_bytes(policy=msg.policy.clone(linesep=os.linesep)))
17
```

Les objets de stratégie peuvent également être combinés à l'aide de l'opérateur d'addition, produisant un objet de stratégie dont les paramètres sont une combinaison des objets additionnées (en ne prenant en compte que les paramètres données explicitement) :

```
>>> compat_SMTP = policy.compat32.clone(linesep='\r\n')
>>> compat_strict = policy.compat32.clone(raise_on_defect=True)
>>> compat_strict_SMTP = compat_SMTP + compat_strict
```

Cette opération n'est pas commutative ; c'est-à-dire que l'ordre dans lequel les objets sont ajoutés est important. Pour illustrer :

```
>>> policy100 = policy.compat32.clone(max_line_length=100)
>>> policy80 = policy.compat32.clone(max_line_length=80)
>>> apolicy = policy100 + policy80
>>> apolicy.max_line_length
80
>>> apolicy = policy80 + policy100
>>> apolicy.max_line_length
100
```

class email.policy.**Policy**(***kw*)

C'est la *classe mère abstraite* pour toutes les classes de définition de politique. Elle fournit des implémentations par défaut pour quelques méthodes triviales, ainsi que l'implémentation de la propriété d'immutabilité, la méthode `clone()` et la sémantique du constructeur.

Le constructeur d'une classe de définition de politique peut recevoir divers arguments nommés. Les arguments qui peuvent être spécifiés sont toutes les propriétés qui ne sont pas des méthodes sur cette classe, plus toutes les propriétés qui ne sont pas des méthodes supplémentaires sur la classe concrète. Une valeur spécifiée dans le constructeur remplace la valeur par défaut de l'attribut correspondant.

Cette classe définit les propriétés suivantes, et ainsi les valeurs des éléments suivants peuvent être passées dans le constructeur de n'importe quelle classe de définition de politique :

max_line_length

La longueur maximale de toute ligne dans la sortie sérialisée, sans compter le(s) caractère(s) de fin de ligne. La valeur par défaut est 78, selon la **RFC 5322**. Une valeur de `0` ou `None` indique qu'aucun retour à la ligne ne doit être fait.

linesep

La chaîne à utiliser pour terminer les lignes dans la sortie sérialisée. La valeur par défaut est `\n` car c'est la définition interne de fin de ligne utilisée par Python, bien que `\r\n` soit requis par les RFC.

cte_type

Contrôle le type d'encodage de contenu qui peut ou doit être utilisé. Les valeurs possibles sont :

`7bit`	toutes les données doivent être « complétement 7 bits » (ASCII uniquement). Cela signifie que si nécessaire, les données seront encodées à l'aide de l'encodage *quoted-printable* ou *base64*.
`8bit`	les données ne sont pas contraintes d'être uniquement sur 7 bits. Les données dans les en-têtes doivent toujours être en ASCII uniquement et seront donc encodées (voir `fold_binary()` et `utf8` ci-dessous pour les exceptions), mais les parties du corps peuvent utiliser le `8bit` CTE.

Une valeur `cte_type` de `8bit` ne fonctionne qu'avec `BytesGenerator`, et pas avec `Generator` car les chaînes ne peuvent pas contenir de données binaires. Si un `Generator` fonctionne sous une politique qui spécifie `cte_type=8bit`, il agit comme si `cte_type` était `7bit`.

raise_on_defect

Si `True`, tous les défauts rencontrés sont signalés comme des erreurs. Si `False` (la valeur par défaut), les défauts sont passés à la méthode `register_defect()`.

mangle_from_

Si `True`, les lignes commençant par *"From "* dans le corps sont échappées en mettant un > devant elles. Ce paramètre est utilisé lorsque le message est sérialisé par un générateur. Par défaut, vaut `False`.

Ajouté dans la version 3.5.

message_factory

Une fonction de fabrique pour construire un nouvel objet de message vide. Utilisé par l'analyseur lors de la construction des messages. La valeur par défaut est `None`, `Message` est alors utilisé.

Ajouté dans la version 3.6.

verify_generated_headers

If `True` (the default), the generator will raise `HeaderWriteError` instead of writing a header that is improperly folded or delimited, such that it would be parsed as multiple headers or joined with adjacent data. Such headers can be generated by custom header classes or bugs in the `email` module.

As it's a security feature, this defaults to `True` even in the `Compat32` policy. For backwards compatible, but unsafe, behavior, it must be set to `False` explicitly.

Ajouté dans la version 3.13.

La méthode `Policy` suivante est destinée à être appelée par le code à l'aide de la bibliothèque de messagerie pour créer des instances de stratégie avec des paramètres personnalisés :

clone(***kw*)

Renvoie une nouvelle instance de `Policy` dont les attributs ont les mêmes valeurs que l'instance actuelle, sauf si ces attributs reçoivent de nouvelles valeurs par les arguments nommés.

Les méthodes `Policy` restantes sont appelées par le code du paquet de messagerie et ne sont pas destinées à être appelées par une application utilisant le paquet de messagerie. Une stratégie personnalisée doit implémenter toutes ces méthodes.

handle_defect(*obj*, *defect*)

Handle a *defect* found on *obj*. When the email package calls this method, *defect* will always be a subclass of `MessageDefect`.

L'implémentation par défaut vérifie le drapeau `raise_on_defect`. Si c'est `True`, *defect* est levé comme une exception. Si c'est `False` (par défaut), *obj* et *defect* sont passés à `register_defect()`.

register_defect(*obj*, *defect*)

Register a *defect* on *obj*. In the email package, *defect* will always be a subclass of `MessageDefect`.

L'implémentation par défaut appelle la méthode `append` de l'attribut `defects` de *obj*. Lorsque le paquet de courrier électronique appelle `handle_defect`, *obj* a normalement un attribut `defects` qui a une méthode `append`. Les types d'objets personnalisés utilisés avec le paquet de messagerie (par exemple,

les objets `Message` personnalisés) doivent également fournir un tel attribut, sinon les défauts dans les messages analysés généreront des erreurs inattendues.

header_max_count(*name*)

Renvoie le nombre maximal autorisé d'en-têtes nommés *name*.

Appelé lorsqu'un en-tête est ajouté à un objet `EmailMessage` ou `Message`. Si la valeur renvoyée n'est pas `0` ou `None`, et qu'il y a déjà un certain nombre d'en-têtes avec le nom *name* supérieur ou égal à la valeur renvoyée, une `ValueError` est levée.

Parce que le comportement par défaut de `Message.__setitem__` est d'ajouter la valeur à la liste des en-têtes, il est facile de créer des en-têtes en double sans s'en rendre compte. Cette méthode permet à certains en-têtes d'être limités dans le nombre d'instances de cet en-tête qui peuvent être ajoutées à un `Message` par programme (la limite n'est pas respectée par l'analyseur, qui produit fidèlement autant d'en-têtes qu'il en existe dans le message analysé).

L'implémentation par défaut renvoie `None` pour tous les noms d'en-tête.

header_source_parse(*sourcelines*)

Le paquet de courrier électronique appelle cette méthode avec une liste de chaînes, chaque chaîne se terminant par les caractères de séparation de ligne trouvés dans la source analysée. La première ligne comprend le nom de l'en-tête de champ et le séparateur. Tous les espaces blancs dans la source sont préservés. La méthode doit renvoyer le couple `(name, value)` qui doit être stocké dans le `Message` pour représenter l'en-tête analysé.

Si une implémentation souhaite conserver la compatibilité avec les politiques de paquet de messagerie existantes, *name* doit être le nom préservé de la casse (tous les caractères jusqu'au séparateur `:`), tandis que *value* doit être la valeur dépliée (c.-à-d. tous les caractères de séparation de ligne supprimés, mais les espaces blancs conservés intacts), dépouillés des espaces blancs de tête.

sourcelines peut contenir des données binaires échappées de substitution.

Il n'y a pas d'implémentation par défaut

header_store_parse(*name*, *value*)

Le paquet de courrier électronique appelle cette méthode avec le nom et la valeur fournis par le programme d'application lorsque le programme d'application modifie un `Message` par programme (par opposition à un `Message` créé par un analyseur). La méthode doit renvoyer le couple `(name, value)` qui doit être stocké dans le `Message` pour représenter l'en-tête.

Si une implémentation souhaite conserver la compatibilité avec les politiques existantes du paquet de messagerie, le *name* et la *value* doivent être des chaînes ou des sous-classes de chaînes qui ne modifient pas le contenu des arguments transmis.

Il n'y a pas d'implémentation par défaut

header_fetch_parse(*name*, *value*)

Le paquet de courrier électronique appelle cette méthode avec le *name* et la *value* actuellement stockés dans le `Message` lorsque cet en-tête est demandé par le programme d'application, et tout ce que la méthode renvoie est ce qui est renvoyé à l'application comme valeur de l'en-tête récupéré. Notez qu'il peut y avoir plus d'un en-tête avec le même nom stocké dans le `Message` ; la méthode reçoit le nom et la valeur spécifiques de l'en-tête destiné à être renvoyé à l'application.

value peut contenir des données binaires de substitution échappées. Il ne doit pas y avoir de données binaires échappées de substitution dans la valeur renvoyée par la méthode.

Il n'y a pas d'implémentation par défaut

fold(*name*, *value*)

Le paquet de courrier électronique appelle cette méthode avec le *name* et la *value* actuellement stockés dans le `Message` pour un en-tête donné. La méthode doit renvoyer une chaîne qui représente correctement cet en-tête « plié » (selon les paramètres de la politique) en mettant en forme le couple *name* - *value* et en insérant les caractères `linesep` aux endroits appropriés. Voir **RFC 5322** pour une discussion sur les règles de mise en forme (« pliage » ou *folding* en anglais) des en-têtes de courrier électronique.

value peut contenir des données binaires de substitution échappées. Il ne doit pas y avoir de données binaires échappées de substitution dans la chaîne renvoyée par la méthode.

fold_binary(*name*, *value*)

Identique à `fold()`, sauf que la valeur renvoyée doit être un objet bytes plutôt qu'une chaîne.

value peut contenir des données binaires de substitution échappées. Celles-ci pourraient être reconverties en données binaires dans l'objet bytes renvoyé.

class email.policy.**EmailPolicy**(**kw)

Cette `Policy` concrète fournit un comportement destiné à être entièrement conforme aux RFC de messagerie actuelles. Ce qui inclut (mais n'est pas limité à) la **RFC 5322**, la **RFC 2047** et les RFC MIME actuelles.

Cette politique ajoute de nouveaux algorithmes d'analyse d'en-tête et de mise en forme (« pliage »). Au lieu de simples chaînes, les en-têtes sont des sous-classes `str` avec des attributs qui dépendent du type du champ. L'algorithme d'analyse et de pliage implémente entièrement la **RFC 2047** et la **RFC 5322**.

La valeur par défaut de l'attribut `message_factory` est `EmailMessage`.

Outre les attributs définissables répertoriés ci-dessus qui s'appliquent à toutes les règles, cette règle ajoute les attributs supplémentaires suivants :

Ajouté dans la version 3.6 : [1]

utf8

Si c'est `False`, suit la **RFC 5322**, en prenant en charge les caractères non-ASCII dans les en-têtes en les encodant comme des « mots encodés ». Si c'est `True`, suit la **RFC 6532** et utilise l'encodage `utf-8` pour les en-têtes. Les messages formatés de cette manière peuvent être transmis aux serveurs SMTP qui gèrent l'extension `SMTPUTF8` (**RFC 6531**).

refold_source

Si la valeur d'un en-tête dans l'objet `Message` provient d'un `parser` (au lieu d'être définie par un programme), cet attribut indique si un générateur doit ou non remettre en forme cette valeur lorsqu'il retransforme le message sous forme sérialisée. Les valeurs possibles sont :

`none`	toutes les valeurs sources utilisent la mise en forme (*folding* en anglais) d'origine
`long`	les valeurs sources dont une ligne est plus longue que `max_line_length` sont remises en forme
`all`	toutes les valeurs sont remises en forme.

La valeur par défaut est `long`.

header_factory

Appelable qui prend deux arguments, `name` et `value`, où `name` est un nom de champ d'en-tête et `value` est une valeur de champ d'en-tête dépliée (c-à-d. sans retours à la ligne de mise en forme), et renvoie une sous-classe de chaîne qui représente cet en-tête. Un `header_factory` par défaut (voir `headerregistry`) est fourni qui prend en charge l'analyse personnalisée pour les divers types de champs d'en-tête d'adresse et de date **RFC 5322**, et les principaux types de champs d'en-tête MIME. La prise en charge d'une analyse personnalisée supplémentaire sera ajoutée à l'avenir.

content_manager

Objet avec au moins deux méthodes : *get_content* et *set_content*. Lorsque la méthode `get_content()` ou `set_content()` d'un objet `EmailMessage` est appelée, elle appelle la méthode correspondante de cet objet, en lui transmettant l'objet message comme premier argument, et tous les arguments positionnels ou nommés qui lui ont été transmis comme arguments supplémentaires. Par défaut, `content_manager` est défini sur `raw_data_manager`.

Ajouté dans la version 3.4.

La classe fournit les implémentations concrètes suivantes des méthodes abstraites de `Policy` :

1. Initialement ajouté dans 3.3 en tant que *paquet provisoire*.

header_max_count(*name*)

Renvoie la valeur de l'attribut `max_count` de la classe spécialisée utilisée pour représenter l'en-tête avec le nom donné.

header_source_parse(*sourcelines*)

Le nom est analysé comme le reste jusqu'au `:` et renvoyé sans modification. La valeur est déterminée en supprimant les espaces blancs de début du reste de la première ligne, en joignant toutes les lignes suivantes et en supprimant tous les caractères de retour chariot ou de saut de ligne de fin.

header_store_parse(*name*, *value*)

Le nom est renvoyé tel quel. Si la valeur d'entrée a un attribut `name` et qu'elle correspond à *name* sans tenir compte de la casse, la valeur est renvoyée inchangée. Sinon, le *name* et la *value* sont passés à `header_factory`, et l'objet d'en-tête résultant est renvoyé comme valeur. Dans ce cas, une `ValueError` est levée si la valeur d'entrée contient des caractères CR ou LF.

header_fetch_parse(*name*, *value*)

Si la valeur a un attribut `name`, elle est renvoyée sans modification. Sinon, le *name* et la *value* sans les caractères CR ou LF sont passés à `header_factory`, et l'objet d'en-tête résultant est renvoyé. Tous les octets échappés de substitution sont transformés en glyphe Unicode de caractère inconnu.

fold(*name*, *value*)

La mise en forme d'en-tête est contrôlée par le paramètre de politique `refold_source`. Une valeur est considérée comme une « valeur source » si et seulement si elle n'a pas d'attribut `name` (avoir un attribut `name` signifie qu'il s'agit d'un objet d'en-tête quelconque). Si une valeur source doit être mise en forme conformément à la politique, elle est convertie en un objet d'en-tête en passant le *name* et la *value* avec tous les caractères CR et LF supprimés à `header_factory`. La mise en forme multi-lignes d'un objet d'en-tête (« pliage » ou *folding* en anglais) se fait en appelant sa méthode `fold` avec la politique actuelle.

Les valeurs sources sont divisées en lignes en utilisant `splitlines()`. Si la valeur ne doit pas être remise en forme, les lignes sont jointes en utilisant le caractère `linesep` de la politique et renvoyées. L'exception concerne les lignes contenant des données binaires non ASCII. Dans ce cas, la valeur est remise en forme quel que soit le paramètre `refold_source`, ce qui entraîne le codage CTE des données binaires à l'aide du jeu de caractères `unknown-8bit`.

fold_binary(*name*, *value*)

Identique à `fold()` si `cte_type` vaut `7bit`, sauf que la valeur renvoyée est de type *bytes*.

Si `cte_type` est `8bit`, les données binaires non-ASCII sont reconverties en octets. Les en-têtes contenant des données binaires ne sont pas remises en forme, quel que soit le paramètre `refold_header`, puisqu'il n'y a aucun moyen de savoir si les données binaires sont constituées de caractères à un seul octet ou de caractères à plusieurs octets.

Les instances suivantes de `EmailPolicy` fournissent des valeurs par défaut adaptées à des domaines d'application spécifiques. Notez qu'à l'avenir, le comportement de ces instances (en particulier l'instance `HTTP`) pourra être ajusté pour se conformer encore plus étroitement aux RFC relatives à leurs domaines.

email.policy.**default**

Une instance de `EmailPolicy` avec toutes les valeurs par défaut inchangées. Cette politique utilise les fins de ligne standard Python `\n` plutôt que le `\r\n` conforme à la RFC.

email.policy.**SMTP**

Convient pour la sérialisation des messages conformément aux RFC des e-mails. Comme `default`, mais avec `linesep` défini sur `\r\n`, ce qui est conforme à la RFC.

email.policy.**SMTPUTF8**

Identique à `SMTP` sauf que `utf8` vaut `True`. Utile pour sérialiser des messages dans une banque de messages sans utiliser de mots encodés dans les en-têtes. Ne doit être utilisée pour la transmission SMTP que si les adresses de l'expéditeur ou du destinataire contiennent des caractères non ASCII (la méthode `smtplib.SMTP.send_message()` gère cela automatiquement).

email.policy.**HTTP**

Convient pour la sérialisation des en-têtes à utiliser dans le trafic HTTP. Comme `SMTP` sauf que `max_line_length` est défini sur `None` (illimité).

email.policy.**strict**

Exemple de commodité. Identique à `default` sauf que `raise_on_defect` est défini sur `True`. Cela permet de rendre stricte toute politique en écrivant :

```
somepolicy + policy.strict
```

Avec toutes ces *EmailPolicies*, l'API effective du paquet de messagerie est modifiée par rapport à l'API Python 3.2 de la manière suivante :

— Définir un en-tête sur un *Message* entraîne l'analyse de cet en-tête et la création d'un objet d'en-tête.
— La récupération d'une valeur d'en-tête à partir d'un *Message* entraîne l'analyse de cet en-tête ainsi que la création et le renvoi d'un objet d'en-tête.
— Tout objet d'en-tête, ou tout en-tête qui est remis en forme en raison des paramètres de politique, est mis en forme à l'aide d'un algorithme qui implémente entièrement les algorithmes de pliage de la RFC, notamment en sachant où les mots encodés sont requis et autorisés.

Du point de vue de l'application, cela signifie que tout en-tête obtenu via *EmailMessage* est un objet d'en-tête avec des attributs supplémentaires, dont la valeur de chaîne est la valeur Unicode entièrement décodée de l'en-tête. De même, un en-tête peut se voir attribuer une nouvelle valeur, ou un nouvel en-tête créé, à l'aide d'une chaîne Unicode, et la politique se chargera de convertir la chaîne Unicode dans la forme encodée RFC correcte.

Les objets d'en-tête et leurs attributs sont décrits dans *headerregistry*.

class email.policy.**Compat32**(***kw*)

Cette *Policy* concrète est la politique de rétrocompatibilité. Elle reproduit le comportement du paquet de courrier électronique dans Python 3.2. Le module *policy* définit également une instance de cette classe, *compat32*, qui est utilisée comme politique par défaut. Ainsi, le comportement par défaut du paquet de messagerie est de maintenir la compatibilité avec Python 3.2.

Les attributs suivants ont des valeurs différentes de la valeur par défaut de *Policy* :

mangle_from_

La valeur par défaut est `True`.

La classe fournit les implémentations concrètes suivantes des méthodes abstraites de *Policy* :

header_source_parse(*sourcelines*)

Le nom est analysé comme le reste jusqu'au `:` et renvoyé sans modification. La valeur est déterminée en supprimant les espaces blancs de début du reste de la première ligne, en joignant toutes les lignes suivantes et en supprimant tous les caractères de retour chariot ou de saut de ligne de fin.

header_store_parse(*name*, *value*)

Le nom et la valeur sont renvoyés sans modification.

header_fetch_parse(*name*, *value*)

Si la valeur contient des données binaires, elle est convertie en un objet *Header* en utilisant le jeu de caractères `unknown-8bit`. Sinon, elle est renvoyée sans modification.

fold(*name*, *value*)

Les en-têtes sont mis en forme à l'aide de l'algorithme de pliage *Header*, qui préserve les sauts de ligne existants dans la valeur et encapsule chaque ligne résultante dans la `max_line_length`. Les données binaires non-ASCII sont codées en CTE à l'aide du jeu de caractères `unknown-8bit`.

fold_binary(*name*, *value*)

Les en-têtes sont mis en forme à l'aide de l'algorithme de pliage *Header*, qui préserve les sauts de ligne existants dans la valeur et encapsule chaque ligne résultante dans la `max_line_length`. Si `cte_type` est `7bit`, les données binaires non-ASCII sont encodées en CTE en utilisant le jeu de caractères `unknown-8bit`. Sinon, l'en-tête source d'origine est utilisé, avec ses sauts de ligne existants et toutes les données binaires (non conformes à la RFC) qu'il peut contenir.

email.policy.**compat32**

Instance de *Compat32*, offrant une rétrocompatibilité avec le comportement du paquet de messagerie dans Python 3.2.

Notes

20.1.5 email.errors : Exception and Defect classes

Code source : Lib/email/errors.py

Les classes d'exception suivantes sont définies dans le module *email.errors* :

exception email.errors.**MessageError**

Exception de base, dont héritent toutes les exceptions du paquet *email*. Cette classe hérite de la classe native *Exception* et ne définit aucune méthode additionnelle.

exception email.errors.**MessageParseError**

Exception de base pour les exceptions levées par la classe *Parser*. Elle hérite de *MessageError*. Cette classe est aussi utilisée en interne par l'analyseur de *headerregistry*.

exception email.errors.**HeaderParseError**

Cette exception, dérivée de *MessageParseError*, est levée sous différentes conditions lors de l'analyse des en-têtes **RFC 5322** du message. Lorsque la méthode *set_boundary()* est invoquée, elle lève cette erreur si le type du contenu est inconnu. La classe *Header* lève cette exception pour certains types d'erreurs provenant du décodage base64. Elle la lève aussi quand un en-tête est créé et qu'il semble contenir un en-tête imbriqué, c'est-à-dire que la ligne qui suit ressemble à un en-tête et ne commence pas par des caractères d'espacement.

exception email.errors.**BoundaryError**

Obsolète, n'est plus utilisé.

exception email.errors.**MultipartConversionError**

Cette exception est levée quand le contenu, que la méthode `add_payload()` essaie d'ajouter à l'objet *Message*, est déjà un scalaire et que le type principal du message *Content-Type* est manquant ou différent de *multipart*. *MultipartConversionError* hérite à la fois de *MessageError* et de *TypeError*.

Comme la méthode `Message.add_payload()` est obsolète, cette exception est rarement utilisée. Néanmoins, elle peut être levée si la méthode *attach()* est invoquée sur une instance de classe dérivée de *MIMENonMultipart* (p. ex. *MIMEImage*).

exception email.errors.**HeaderWriteError**

Raised when an error occurs when the *generator* outputs headers.

exception email.errors.**MessageDefect**

This is the base class for all defects found when parsing email messages. It is derived from *ValueError*.

exception email.errors.**HeaderDefect**

This is the base class for all defects found when parsing email headers. It is derived from *MessageDefect*.

Voici la liste des anomalies que peut identifier *FeedParser* pendant l'analyse des messages. Notez que les anomalies sont signalées à l'endroit où elles sont détectées : par exemple, dans le cas d'une malformation de l'en-tête d'un message imbriqué dans un message de type *multipart/alternative*, l'anomalie est signalée sur le message imbriqué seulement.

Toutes les anomalies sont des sous-classes de *email.errors.MessageDefect*.

— `NoBoundaryInMultipartDefect` — Un message qui prétend être composite (*multipart* en anglais), mais qui ne contient pas de séparateur *boundary*.

— `StartBoundaryNotFoundDefect` — Le message ne contient pas le séparateur de départ indiqué dans le *Content-Type*.

— `CloseBoundaryNotFoundDefect` — Le séparateur de départ a été trouvé, mais pas le séparateur de fin correspondant.

Ajouté dans la version 3.3.

— `FirstHeaderLineIsContinuationDefect` — La première ligne de l'en-tête du message est une ligne de continuation.
— `MisplacedEnvelopeHeaderDefect` — Un en-tête *Unix From* est présent à l'intérieur d'un bloc d'en-tête.
— `MissingHeaderBodySeparatorDefect` — Une ligne d'en-tête ne contient pas de caractères d'espacement au début et aucun « : ». L'analyse continue en supposant qu'il s'agit donc de la première ligne du corps du message.

Ajouté dans la version 3.3.

— `MalformedHeaderDefect` -- Un en-tête est mal formé ou il manque un « : ».

Obsolète depuis la version 3.3 : Cette anomalie est obsolète depuis plusieurs versions de Python.

— `MultipartInvariantViolationDefect` — Le message indique être de type `multipart`, mais aucune pièce jointe n'a été trouvée. Notez que, dans ce cas, la méthode `is_multipart()` peut renvoyer `False` même si le type de contenu est `multipart`.
— `InvalidBase64PaddingDefect` — Remplissage incorrect d'un bloc d'octets encodés en base64. Des caractères de remplissage ont été ajoutés pour permettre le décodage, mais le résultat du décodage peut être invalide.
— `InvalidBase64CharactersDefect` — Des caractères n'appartenant pas à l'alphabet base64 ont été rencontrés lors du décodage d'un bloc d'octets encodés en base64. Les caractères ont été ignorés, mais le résultat du décodage peut être invalide.
— `InvalidBase64LengthDefect` — Le nombre de caractères (autres que de remplissage) d'un bloc d'octets encodés en base64 est invalide (1 de plus qu'un multiple de 4). Le bloc encodé n'a pas été modifié.
— `InvalidDateDefect` — Le champ de date est invalide. La valeur est laissée telle-quelle.

20.1.6 `email.headerregistry` : Custom Header Objects

Code source : Lib/email/headerregistry.py

Ajouté dans la version 3.6 : [1]

Les en-têtes sont représentés par des sous-classes personnalisées de `str`. La classe particulière utilisée pour représenter un en-tête donné est déterminée par `header_factory` de la `policy` en vigueur lorsque les en-têtes sont créés. Cette section documente la `header_factory` particulière implémentée par le paquet de messagerie pour la gestion des messages électroniques conformes à la **RFC 5322**, qui fournit non seulement des objets d'en-tête personnalisés pour différents types d'en-tête, mais fournit également un mécanisme d'extension permettant aux applications d'ajouter leurs propres types d'en-tête personnalisés.

Lors de l'utilisation de l'un des objets de définition de politique dérivés de `EmailPolicy`, tous les en-têtes sont produits par `HeaderRegistry` et ont `BaseHeader` comme dernière classe mère. Chaque classe d'en-tête a une classe mère supplémentaire qui est déterminée par le type de l'en-tête. Par exemple, de nombreux en-têtes ont la classe `UnstructuredHeader` comme autre classe mère. La deuxième classe spécialisée pour un en-tête est déterminée par le nom de l'en-tête, en utilisant un tableau de recherche stocké dans `HeaderRegistry`. Tout cela est géré de manière transparente pour le programme d'application typique, mais des interfaces sont fournies pour modifier le comportement par défaut pour une utilisation par des applications plus complexes.

Les sections ci-dessous documentent d'abord les classes mères d'en-tête et leurs attributs, suivies de l'API pour modifier le comportement de `HeaderRegistry`, et enfin les classes de gestion utilisées pour représenter les données analysées à partir d'en-têtes structurés.

class `email.headerregistry.`**`BaseHeader`**(*name*, *value*)

name et *value* sont passés à `BaseHeader` à partir de l'appel `header_factory`. La valeur de chaîne de tout objet d'en-tête est la *value* entièrement décodée en Unicode.

Cette classe mère définit les propriétés en lecture seule suivantes :

name

Nom de l'en-tête (la partie du champ avant le `:`). C'est exactement la valeur passée dans l'appel `header_factory` pour *name* ; c'est-à-dire que la casse est préservée.

1. Ajouté à l'origine dans 3.3 en tant que *paquet provisoire*

defects

n-uplet d'instances `HeaderDefect` signalant tout problème de conformité avec les RFC détecté lors de l'analyse. Le paquet d'e-mails tente d'être complet en ce qui concerne la détection des problèmes de conformité. Voir le module `errors` pour une discussion sur les types de défauts qui peuvent être signalés.

max_count

Nombre maximum d'en-têtes de ce type pouvant avoir le même `name`. Une valeur de `None` signifie illimité. La valeur `BaseHeader` pour cet attribut est `None` ; les classes d'en-tête spécialisées sont censées remplacer cette valeur si nécessaire.

`BaseHeader` fournit également la méthode suivante, qui est appelée par le code de la bibliothèque de messagerie et ne doit généralement pas être appelée par les programmes d'application :

fold(*, *policy*)

Renvoie une chaîne contenant les caractères `linesep` nécessaires pour remettre en forme correctement l'en-tête conformément à *policy*. Un `cte_type` de `8bit` est traité comme s'il s'agissait de `7bit`, car les en-têtes ne peuvent pas contenir de données binaires arbitraires. Si `utf8` est `False`, les données non-ASCII seront encodées selon la **RFC 2047**.

`BaseHeader` seule ne peut pas être utilisée pour créer un objet d'en-tête. Elle définit un protocole avec lequel chaque en-tête spécialisé coopère afin de produire l'objet d'en-tête. Plus précisément, `BaseHeader` nécessite que la classe spécialisée fournisse une `classmethod()` nommée `parse`. Cette méthode s'appelle comme suit :

```
parse(string, kwds)
```

`kwds` est un dictionnaire contenant une clé pré-initialisée : `defects`. `defects` est une liste vide. La méthode d'analyse doit ajouter tous les défauts détectés à cette liste. Au retour, le dictionnaire `kwds` *doit* contenir des valeurs pour au moins les clés `decoded` et `defects`. `decoded` doit être la valeur de la chaîne pour l'en-tête (c'est-à-dire la valeur de l'en-tête entièrement décodée en Unicode). La méthode d'analyse doit supposer que *string* peut contenir des parties codées par transfert de contenu, mais doit également gérer correctement tous les caractères Unicode valides afin de pouvoir analyser les valeurs d'en-tête non codées.

`__new__` de `BaseHeader` crée ensuite l'instance d'en-tête et appelle sa méthode `init`. La classe spécialisée n'a besoin de fournir une méthode `init` que si elle souhaite définir des attributs supplémentaires au-delà de ceux fournis par `BaseHeader` lui-même. Une telle méthode `init` devrait ressembler à ceci :

```
def init(self, /, *args, **kw):
    self._myattr = kw.pop('myattr')
    super().init(*args, **kw)
```

Autrement dit, tout ce que la classe spécialisée ajoute au dictionnaire `kwds` doit être supprimé et géré, et le contenu restant de `kw` (et `args`) est passé à la méthode `init` de `BaseHeader`.

class email.headerregistry.**UnstructuredHeader**

Un en-tête « non structuré » est le type d'en-tête par défaut dans la **RFC 5322**. Tout en-tête qui n'a pas de syntaxe spécifiée est traité comme non structuré. L'exemple classique d'en-tête non structuré est l'en-tête `Subject`.

Dans la **RFC 5322**, un en-tête non structuré est une séquence de texte arbitraire dans le jeu de caractères ASCII. **RFC 2047**, cependant, possède un mécanisme compatible **RFC 5322** pour encoder du texte non-ASCII en tant que caractères ASCII dans une valeur d'en-tête. Lorsqu'une *value* contenant des mots encodés est passée au constructeur, l'analyseur `UnstructuredHeader` convertit ces mots encodés en Unicode, en suivant les règles de la **RFC 2047** pour le texte non structuré. L'analyseur utilise des heuristiques pour tenter de décoder certains mots codés non conformes. Des défauts sont enregistrés dans de tels cas, ainsi que des défauts pour des problèmes tels que des caractères non valides dans les mots codés ou le texte non codé.

Ce type d'en-tête ne fournit aucun attribut supplémentaire.

class email.headerregistry.**DateHeader**

La **RFC 5322** spécifie un format très spécifique pour les dates dans les en-têtes de courrier électronique. L'analyseur `DateHeader` reconnaît ce format de date, ainsi qu'un certain nombre de formes variantes que l'on trouve parfois « dans la nature ».

Ce type d'en-tête fournit les attributs supplémentaires suivants :

datetime

Si la valeur d'en-tête peut être reconnue comme une date valide d'une forme ou d'une autre, cet attribut contient une instance *datetime* représentant cette date. Si le fuseau horaire de la date d'entrée est spécifié comme `-0000` (indiquant qu'il est en UTC mais ne contient aucune information sur le fuseau horaire source), alors *datetime* est une *datetime* naïve. Si un décalage de fuseau horaire spécifique est trouvé (y compris `+0000`), alors *datetime* contient un `datetime` avisé qui utilise *datetime.timezone* pour enregistrer le décalage lié au fuseau horaire.

La valeur `decoded` de l'en-tête est déterminée en formatant le `datetime` selon les règles de la **RFC 5322** ; c'est-à-dire qu'elle est définie sur :

```
email.utils.format_datetime(self.datetime)
```

Lors de la création d'un `DateHeader`, *value* peut être une instance de *datetime*. Cela signifie, par exemple, que le code suivant est valide et fait ce à quoi on pourrait s'attendre :

```
msg['Date'] = datetime(2011, 7, 15, 21)
```

Comme il s'agit d'un `datetime` naïf, il est interprété comme un horodatage UTC et la valeur résultante a un fuseau horaire de `-0000`. Il est beaucoup plus utile d'utiliser la fonction *localtime()* du module *utils* :

```
msg['Date'] = utils.localtime()
```

Cet exemple définit l'en-tête de date sur l'heure et la date actuelles à l'aide du décalage horaire actuel.

class email.headerregistry.**AddressHeader**

Les en-têtes d'adresse sont l'un des types d'en-têtes structurés les plus complexes. La classe `AddressHeader` fournit une interface générique à n'importe quel en-tête d'adresse.

Ce type d'en-tête fournit les attributs supplémentaires suivants :

groups

n-uplet d'objets *Group* encodant les adresses et les groupes trouvés dans la valeur d'en-tête. Les adresses qui ne font pas partie d'un groupe sont représentées dans cette liste comme des `Groups` à adresse unique dont *display_name* est `None`.

addresses

n-uplet d'objets *Address* encodant toutes les adresses individuelles à partir de la valeur d'en-tête. Si la valeur d'en-tête contient des groupes, les adresses individuelles du groupe sont incluses dans la liste au point où le groupe apparaît dans la valeur (c'est-à-dire que la liste d'adresses est « aplatie » en une liste unidimensionnelle).

La valeur `decoded` de l'en-tête a tous les mots codés décodés en Unicode. Les noms de domaine encodés en *idna* sont également décodés en Unicode. La valeur `decoded` est définie en *joignant* la valeur *str* des éléments de l'attribut `groups` avec `', '`.

Une liste d'objets *Address* et *Group* dans n'importe quelle combinaison peut être utilisée pour définir la valeur d'un en-tête d'adresse. Les objets `Group` dont le `display_name` est `None` sont interprétés comme des adresses uniques, ce qui permet de copier une liste d'adresses avec des groupes intacts en utilisant la liste obtenue à partir de l'attribut `groups` de l'en-tête source.

class email.headerregistry.**SingleAddressHeader**

Sous-classe de *AddressHeader* qui ajoute un attribut supplémentaire :

address

Adresse unique codée par la valeur d'en-tête. Si la valeur d'en-tête contient en fait plus d'une adresse (ce qui serait une violation de la RFC sous la valeur par défaut *policy*), l'accès à cet attribut lève une *ValueError*.

La plupart des classes ci-dessus ont également une variante `Unique` (par exemple, `UniqueUnstructuredHeader`). La seule différence est que dans la variante `Unique`, *max_count* est définie sur 1.

class email.headerregistry.**MIMEVersionHeader**

Il n'y a vraiment qu'une seule valeur valide pour l'en-tête *MIME-Version*, et c'est `1.0`. Au cas où, cette classe d'en-tête prend en charge d'autres numéros de version valides. Si un numéro de version a une valeur valide selon la **RFC 2045**, alors l'objet d'en-tête a des valeurs autres que `None` pour les attributs suivants :

version

Numéro de version sous forme de chaîne, sans espaces ni commentaires.

major

Numéro de version majeure sous forme d'entier

minor

Numéro de version mineure sous forme d'entier

class email.headerregistry.**ParameterizedMIMEHeader**

Les en-têtes MIME commencent tous par le préfixe `'Content-'`. Chaque en-tête spécifique a une certaine valeur, décrite par la classe de cet en-tête. Certains peuvent également prendre une liste de paramètres supplémentaires, qui ont un format commun. Cette classe sert de base à tous les en-têtes MIME qui prennent des paramètres.

params

Dictionnaire de correspondance entre les noms de paramètres et leurs valeurs.

class email.headerregistry.**ContentTypeHeader**

Classe *ParameterizedMIMEHeader* qui gère l'en-tête *Content-Type*.

content_type

Chaîne de type de contenu, sous la forme `maintype/subtype`.

maintype

subtype

class email.headerregistry.**ContentDispositionHeader**

Classe *ParameterizedMIMEHeader* qui gère l'en-tête *Content-Disposition*.

content_disposition

`inline` et `attachment` sont les seules valeurs valides couramment utilisées.

class email.headerregistry.**ContentTransferEncoding**

Gère l'en-tête *Content-Transfer-Encoding*.

cte

Les valeurs valides sont `7bit`, `8bit`, `base64` et `quoted-printable`. Voir la **RFC 2045** pour plus d'informations.

class email.headerregistry.**HeaderRegistry**(*base_class=BaseHeader*, *default_class=UnstructuredHeader*, *use_default_map=True*)

This is the factory used by *EmailPolicy* by default. `HeaderRegistry` builds the class used to create a header instance dynamically, using *base_class* and a specialized class retrieved from a registry that it holds. When a given header name does not appear in the registry, the class specified by *default_class* is used as the specialized class. When *use_default_map* is `True` (the default), the standard mapping of header names to classes is copied in to the registry during initialization. *base_class* is always the last class in the generated class's `__bases__` list.

Les correspondances par défaut sont :

subject
UniqueUnstructuredHeader

date
UniqueDateHeader

resent-date
DateHeader

orig-date
UniqueDateHeader

sender
UniqueSingleAddressHeader

resent-sender
SingleAddressHeader

to
UniqueAddressHeader

resent-to
AddressHeader

cc
UniqueAddressHeader

resent-cc
AddressHeader

bcc
UniqueAddressHeader

resent-bcc
AddressHeader

from
UniqueAddressHeader

resent-from
AddressHeader

reply-to
UniqueAddressHeader

mime-version
MIMEVersionHeader

content-type
ContentTypeHeader

content-disposition
ContentDispositionHeader

content-transfer-encoding
ContentTransferEncodingHeader

message-id
MessageIDHeader

`HeaderRegistry` a les méthodes suivantes :

`map_to_type`(*self*, *name*, *cls*)

name est le nom de l'en-tête à faire correspondre. Il est converti en minuscules dans le registre. *cls* est la classe spécialisée à utiliser, avec *base_class*, pour créer la classe utilisée pour instancier les en-têtes qui correspondent à *name*.

`__getitem__`(*name*)

Construit et renvoie une classe pour gérer la création d'un en-tête *name*.

`__call__`(*name*, *value*)

Récupère l'en-tête spécialisé associé à *name* du registre (en utilisant *default_class* si *name* n'apparaît pas dans le registre) et le compose avec *base_class* pour produire une classe, appelle le constructeur de la classe construite, en lui passant la même liste d'arguments, et renvoie enfin l'instance de classe ainsi créée.

Les classes suivantes sont les classes utilisées pour représenter les données analysées à partir d'en-têtes structurés et peuvent, en général, être utilisées par un programme d'application pour construire des valeurs structurées à affecter à des en-têtes spécifiques.

class email.headerregistry.**Address**(*display_name=''*, *username=''*, *domain=''*, *addr_spec=None*)

Classe utilisée pour représenter une adresse e-mail. La forme générale d'une adresse est :

```
[display_name] <username@domain>
```

ou :

```
username@domain
```

où chaque partie doit se conformer à des règles de syntaxe spécifiques énoncées dans la **RFC 5322**.

Pour plus de commodité, *addr_spec* peut être spécifié à la place de *username* et *domain*, *username* et *domain* sont alors analysés à partir de *addr_spec*. Une *addr_spec* doit être une chaîne correctement entre guillemets RFC ; si ce n'est pas le cas, `Address` lève une erreur. Les caractères Unicode sont autorisés et sont encodés proprement lors de la sérialisation. Cependant, selon les RFC, l'Unicode n'est *pas* autorisé dans la partie nom d'utilisateur de l'adresse.

display_name

Partie du nom d'affichage de l'adresse, le cas échéant, avec toutes les citations supprimées. Si l'adresse n'a pas de nom d'affichage, cet attribut est une chaîne vide.

username

Partie `username` de l'adresse, sans guillemets.

domain

Partie `domain` de l'adresse.

addr_spec

Partie `username@domain` de l'adresse, correctement citée pour une utilisation en tant qu'adresse nue (la deuxième forme illustrée ci-dessus). Cet attribut n'est pas modifiable.

__str__()

La valeur `str` de l'objet est l'adresse entre guillemets selon les règles de la **RFC 5322**, mais sans encodage de contenu pour les caractères non-ASCII.

Pour prendre en charge SMTP (**RFC 5321**), `Address` gère un cas particulier : si `username` et `domain` sont tous deux la chaîne vide (ou `None`), alors la valeur de chaîne de `Address` est `<>`.

class email.headerregistry.**Group**(*display_name=None*, *addresses=None*)

Classe utilisée pour représenter un groupe d'adresses. La forme générale d'un groupe d'adresses est :

```
display_name: [address-list];
```

Pour faciliter le traitement des listes d'adresses composées d'un mélange de groupes et d'adresses uniques, un `Group` peut également être utilisé pour représenter des adresses uniques qui ne font pas partie d'un groupe en définissant *display_name* sur `None` et en fournissant une liste de l'adresse unique sous la forme *adresses*.

display_name

`display_name` du groupe. Si c'est `None` et qu'il y a exactement une `Adress` dans `adresses`, alors le `Group` représente une seule adresse qui n'est pas dans un groupe.

addresses

n-uplet éventuellement vide d'objets `Address` représentant les adresses du groupe.

__str__()

La valeur `str` d'un `Group` est formatée selon la **RFC 5322**, mais sans encodage de contenu pour les caractères non-ASCII. Si `display_name` est `None` et qu'il y a une seule `Address` dans la liste `addresses`, la valeur `str` est la même que la `str` de cette seule `Adress`.

Notes

20.1.7 email.contentmanager : Managing MIME Content

Code source : Lib/email/contentmanager.py

Ajouté dans la version 3.6 : [1]

class email.contentmanager.**ContentManager**

Classe mère pour les gestionnaires de contenu. Fournit les mécanismes de registre standard pour enregistrer les convertisseurs entre le contenu MIME et d'autres représentations, ainsi que les méthodes de répartition `get_content` et `set_content`.

get_content(*msg*, **args*, ***kw*)

Recherche une fonction de gestion en se basant sur le `mimetype` de *msg* (voir le paragraphe suivant), l'appelle en lui passant tous les arguments et renvoie le résultat de l'appel. Le gestionnaire doit extraire la charge utile de *msg* et renvoyer un objet qui encode les informations relatives aux données extraites.

Pour trouver le gestionnaire, recherche les clés suivantes dans le registre, en s'arrêtant à la première trouvée :
— une chaîne représentant le type MIME complet (`maintype/subtype`) ;
— une chaîne représentant le `maintype` ;
— une chaîne vide.
Si aucune de ces clés ne produit de gestionnaire, lève une `KeyError` pour le type MIME complet.

set_content(*msg*, *obj*, **args*, ***kw*)

Si le `maintype` est `multipart`, lève une `TypeError` ; sinon recherche une fonction de gestion en se basant sur le type de *obj* (voir le paragraphe suivant), appelle `clear_content()` sur le *msg* et appelle la fonction de gestion en lui passant tous les arguments. Le gestionnaire doit transformer et stocker *obj* en *msg*, apportant éventuellement d'autres modifications à *msg* également, comme l'ajout de divers en-têtes MIME pour coder les informations nécessaires à l'interprétation des données stockées.

Pour trouver le gestionnaire, récupère le type de *obj* (`typ = type(obj)`) et recherche les clés suivantes dans le registre, en s'arrêtant à la première trouvée :
— le type lui-même (`typ`) ;
— le nom complet du type (`typ.__module__ + '.' + typ.__qualname__`) ;
— the type's `qualname` (`typ.__qualname__`)
— the type's `name` (`typ.__name__`).
If none of the above match, repeat all of the checks above for each of the types in the *MRO* (`typ.__mro__`). Finally, if no other key yields a handler, check for a handler for the key `None`. If there is no handler for `None`, raise a `KeyError` for the fully qualified name of the type.

Ajoute également un en-tête `MIME-Version` s'il n'y en a pas (voir aussi `MIMEPart`).

add_get_handler(*key*, *handler*)

Enregistre la fonction *handler* comme gestionnaire pour *key*. Pour les valeurs possibles de *key*, voir `get_content()`.

add_set_handler(*typekey*, *handler*)

Enregistre *handler* comme fonction à appeler lorsqu'un objet d'un type correspondant à *typekey* est passé à `set_content()`. Pour les valeurs possibles de *typekey*, voir `set_content()`.

Instances de gestionnaires de contenus

Actuellement, le paquet *email* ne fournit qu'un seul gestionnaire de contenu concret, `raw_data_manager`, bien que d'autres puissent être ajoutés à l'avenir. `raw_data_manager` est le `content_manager` fourni par `EmailPolicy` et ses dérivés.

1. Initialement ajouté dans 3.4 en tant que *paquet provisoire*.

email.contentmanager.**raw_data_manager**

Ce gestionnaire de contenu ne fournit qu'une interface minimale au-delà de celle fournie par `Message` lui-même : il prend seulement en charge le texte, les chaînes d'octets brutes et les objets `Message`. Néanmoins, il offre des avantages significatifs par rapport à l'API de base : `get_content` sur une partie de texte renvoie une chaîne Unicode sans que l'application ait besoin de la décoder manuellement, `set_content` fournit de nombreuses options pour contrôler les en-têtes ajoutés à une partie et l'encodage du transfert de contenu, et il permet l'utilisation des différentes méthodes `add_`, ce qui simplifie la création de messages en plusieurs parties.

email.contentmanager.**get_content**(*msg*, *errors='replace'*)

Renvoie la charge utile de la partie sous la forme d'une chaîne (pour les parties `text`), d'un objet `EmailMessage` (pour les parties `message/rfc822`) ou d'un objet `bytes` (pour tous les autres types à l'exception de *multipart*). Lève une `KeyError` si cette méthode est appelée sur un `multipart`. S'il s'agit d'une partie `text` et que *errors* est spécifié, ce paramètre est utilisé comme gestionnaire d'erreurs lors du décodage de la charge utile en Unicode. Le gestionnaire d'erreurs par défaut est `replace`.

email.contentmanager.**set_content**(*msg*, *<'str'>*, *subtype="plain"*, *charset='utf-8'*, *cte=None*, *disposition=None*, *filename=None*, *cid=None*, *params=None*, *headers=None*)

email.contentmanager.**set_content**(*msg*, *<'bytes'>*, *maintype*, *subtype*, *cte="base64"*, *disposition=None*, *filename=None*, *cid=None*, *params=None*, *headers=None*)

email.contentmanager.**set_content**(*msg*, *<'EmailMessage'>*, *cte=None*, *disposition=None*, *filename=None*, *cid=None*, *params=None*, *headers=None*)

Ajoute des en-têtes et une charge utile à *msg* :

Ajoute un en-tête `Content-Type` avec une valeur `maintype/subtype`.

— Pour `str`, définit le `maintype` MIME à `text` et définit le sous-type à *subtype* s'il est spécifié, ou à `plain` s'il ne l'est pas.
— Pour `bytes`, utilise les *maintype* et *subtype* spécifiés, ou lève une `TypeError` s'ils ne sont pas spécifiés.
— Pour les objets `EmailMessage`, définit le type principal (*maintype*) à `message` et définit le sous-type à *subtype* s'il est spécifié ou à `rfc822` s'il ne l'est pas. Si *subtype* est `partial`, lève une erreur (les objets `bytes` doivent être utilisés pour construire les parties `message/partial`).

Si *charset* est fourni (qui n'est valide que pour `str`), encode la chaîne en octets en utilisant le jeu de caractères spécifié. La valeur par défaut est `utf-8`. Si le *charset* spécifié est un alias connu pour un nom de jeu de caractères MIME standard, utilise plutôt le jeu de caractères standard.

Si *cte* est défini, encode la charge utile à l'aide de l'encodage de transfert de contenu spécifié et définit l'en-tête `Content-Transfer-Encoding` à cette valeur. Les valeurs possibles pour *cte* sont `quoted-printable`, `base64`, `7bit`, `8bit` et `binary`. Si l'entrée ne peut pas être encodée dans l'encodage spécifié (par exemple, en spécifiant un *cte* de `7bit` pour une entrée qui contient des valeurs non-ASCII), lève une `ValueError`.

— For `str` objects, if *cte* is not set use heuristics to determine the most compact encoding. Prior to encoding, `str.splitlines()` is used to normalize all line boundaries, ensuring that each line of the payload is terminated by the current policy's `linesep` property (even if the original string did not end with one).
— For `bytes` objects, *cte* is taken to be base64 if not set, and the aforementioned newline translation is not performed.
— Pour `EmailMessage`, selon la **RFC 2046**, pour le *subtype* `rfc822`, lève une erreur pour un *cte* valant `quoted-printable` ou `base64`. Pour le *subtype* `external-body`, lève une erreur pour tout *cte* autre que `7bit`. Pour `message/rfc822`, la valeur par défaut de *cte* est `8bit`. Pour toutes les autres valeurs de *sous-type*, la valeur par défaut de *cte* est `7bit`.

> **Note**
>
> la valeur `binary` pour *cte* ne fonctionne pas encore correctement. L'objet `EmailMessage` tel que modifié par `set_content` est correct, mais `BytesGenerator` ne le sérialise pas correctement.

Si *disposition* est spécifié, utilise cette valeur pour l'en-tête `Content-Disposition`. S'il n'est pas spécifié et que *filename* est spécifié, utilise la valeur `attachment` pour l'en-tête. Si ni *disposition* ni *filename* ne sont spécifiés, n'ajoute pas cet en-tête. Les seules valeurs valides pour *disposition* sont `attachment` et `inline`.

Si *filename* est spécifié, utilise cette valeur pour le paramètre `filename` de l'en-tête `Content-Disposition`.

Si *cid* est spécifié, ajoute un en-tête `Content-ID` avec cette valeur.

Si *params* est spécifié, itère sur sa méthode `items` et utilise les paires `(clé, valeur)` résultantes pour définir des paramètres supplémentaires dans l'en-tête `Content-Type`.

Si *headers* est spécifié et est une liste de chaînes de la forme `headername: headervalue` ou une liste d'objets `header` (distingués des chaînes par l'attribut `name`), ajoute ces en-têtes à *msg*.

Notes

20.1.8 `email` : Exemples

Cette page contient quelques exemples de l'utilisation du package `email` pour lire, écrire, et envoyer de simples messages mail, ainsi que des messages MIME plus complexes.

Premièrement, regardons comment créer et envoyer un message avec simplement du texte (le contenu textuel et les adresses peuvent tous deux contenir des caractères Unicodes) :

```
# Import smtplib for the actual sending function
import smtplib

# Import the email modules we'll need
from email.message import EmailMessage

# Open the plain text file whose name is in textfile for reading.
with open(textfile) as fp:
    # Create a text/plain message
    msg = EmailMessage()
    msg.set_content(fp.read())

# me == the sender's email address
# you == the recipient's email address
msg['Subject'] = f'The contents of {textfile}'
msg['From'] = me
msg['To'] = you

# Send the message via our own SMTP server.
s = smtplib.SMTP('localhost')
s.send_message(msg)
s.quit()
```

Analyser des entêtes **RFC 822** peut être aisément réalisé en utilisant les classes du module `parser` :

```
# Import the email modules we'll need
#from email.parser import BytesParser
from email.parser import Parser
from email.policy import default

# If the e-mail headers are in a file, uncomment these two lines:
# with open(messagefile, 'rb') as fp:
#     headers = BytesParser(policy=default).parse(fp)

#  Or for parsing headers in a string (this is an uncommon operation), use:
```

(suite sur la page suivante)

(suite de la page précédente)

```
headers = Parser(policy=default).parsestr(
        'From: Foo Bar <user@example.com>\n'
        'To: <someone_else@example.com>\n'
        'Subject: Test message\n'
        '\n'
        'Body would go here\n')

#  Now the header items can be accessed as a dictionary:
print('To: {}'.format(headers['to']))
print('From: {}'.format(headers['from']))
print('Subject: {}'.format(headers['subject']))

# You can also access the parts of the addresses:
print('Recipient username: {}'.format(headers['to'].addresses[0].username))
print('Sender name: {}'.format(headers['from'].addresses[0].display_name))
```

Voici un exemple de l'envoi d'un message MIME contenant une série de photos de famille qui sont stockés ensemble dans un dossier :

```
# Import smtplib for the actual sending function.
import smtplib

# Here are the email package modules we'll need.
from email.message import EmailMessage

# Create the container email message.
msg = EmailMessage()
msg['Subject'] = 'Our family reunion'
# me == the sender's email address
# family = the list of all recipients' email addresses
msg['From'] = me
msg['To'] = ', '.join(family)
msg.preamble = 'You will not see this in a MIME-aware mail reader.\n'

# Open the files in binary mode.  You can also omit the subtype
# if you want MIMEImage to guess it.
for file in pngfiles:
    with open(file, 'rb') as fp:
        img_data = fp.read()
    msg.add_attachment(img_data, maintype='image',
                                 subtype='png')

# Send the email via our own SMTP server.
with smtplib.SMTP('localhost') as s:
    s.send_message(msg)
```

Voici un exemple d'envoi du contenu d'un dossier entier en tant que message mail : [1]

```
#!/usr/bin/env python3

"""Send the contents of a directory as a MIME message."""

import os
import smtplib
# For guessing MIME type based on file name extension
```

(suite sur la page suivante)

1. Merci à Matthew Dixon Cowles pour l'inspiration originale et les exemples.

(suite de la page précédente)

```
import mimetypes

from argparse import ArgumentParser

from email.message import EmailMessage
from email.policy import SMTP


def main():
    parser = ArgumentParser(description="""\
Send the contents of a directory as a MIME message.
Unless the -o option is given, the email is sent by forwarding to your local
SMTP server, which then does the normal delivery process.  Your local machine
must be running an SMTP server.
""")
    parser.add_argument('-d', '--directory',
                        help="""Mail the contents of the specified directory,
                        otherwise use the current directory.  Only the regular
                        files in the directory are sent, and we don't recurse to
                        subdirectories.""")
    parser.add_argument('-o', '--output',
                        metavar='FILE',
                        help="""Print the composed message to FILE instead of
                        sending the message to the SMTP server.""")
    parser.add_argument('-s', '--sender', required=True,
                        help='The value of the From: header (required)')
    parser.add_argument('-r', '--recipient', required=True,
                        action='append', metavar='RECIPIENT',
                        default=[], dest='recipients',
                        help='A To: header value (at least one required)')
    args = parser.parse_args()
    directory = args.directory
    if not directory:
        directory = '.'
    # Create the message
    msg = EmailMessage()
    msg['Subject'] = f'Contents of directory {os.path.abspath(directory)}'
    msg['To'] = ', '.join(args.recipients)
    msg['From'] = args.sender
    msg.preamble = 'You will not see this in a MIME-aware mail reader.\n'

    for filename in os.listdir(directory):
        path = os.path.join(directory, filename)
        if not os.path.isfile(path):
            continue
        # Guess the content type based on the file's extension.  Encoding
        # will be ignored, although we should check for simple things like
        # gzip'd or compressed files.
        ctype, encoding = mimetypes.guess_file_type(path)
        if ctype is None or encoding is not None:
            # No guess could be made, or the file is encoded (compressed), so
            # use a generic bag-of-bits type.
            ctype = 'application/octet-stream'
        maintype, subtype = ctype.split('/', 1)
        with open(path, 'rb') as fp:
            msg.add_attachment(fp.read(),
```

(suite sur la page suivante)

(suite de la page précédente)

```
                                maintype=maintype,
                                subtype=subtype,
                                filename=filename)
    # Now send or store the message
    if args.output:
        with open(args.output, 'wb') as fp:
            fp.write(msg.as_bytes(policy=SMTP))
    else:
        with smtplib.SMTP('localhost') as s:
            s.send_message(msg)


if __name__ == '__main__':
    main()
```

Voici un exemple de comment décomposer un message MIME comme celui ci-dessus en tant que fichiers dans un dossier :

```
#!/usr/bin/env python3

"""Unpack a MIME message into a directory of files."""

import os
import email
import mimetypes

from email.policy import default

from argparse import ArgumentParser


def main():
    parser = ArgumentParser(description="""\
Unpack a MIME message into a directory of files.
""")
    parser.add_argument('-d', '--directory', required=True,
                        help="""Unpack the MIME message into the named
                        directory, which will be created if it doesn't already
                        exist.""")
    parser.add_argument('msgfile')
    args = parser.parse_args()

    with open(args.msgfile, 'rb') as fp:
        msg = email.message_from_binary_file(fp, policy=default)

    try:
        os.mkdir(args.directory)
    except FileExistsError:
        pass

    counter = 1
    for part in msg.walk():
        # multipart/* are just containers
        if part.get_content_maintype() == 'multipart':
            continue
        # Applications should really sanitize the given filename so that an
```

(suite sur la page suivante)

(suite de la page précédente)

```
        # email message can't be used to overwrite important files
        filename = part.get_filename()
        if not filename:
            ext = mimetypes.guess_extension(part.get_content_type())
            if not ext:
                # Use a generic bag-of-bits extension
                ext = '.bin'
            filename = f'part-{counter:03d}{ext}'
        counter += 1
        with open(os.path.join(args.directory, filename), 'wb') as fp:
            fp.write(part.get_payload(decode=True))


if __name__ == '__main__':
    main()
```

Voici un exemple de création d'un message HTML avec une version en texte comme alternative. Pour rendre les choses un peu plus intéressantes, nous incluons aussi une image dans la partie HTML, nous sauvons une copie du message sur le disque, et nous l'envoyons.

```
#!/usr/bin/env python3

import smtplib

from email.message import EmailMessage
from email.headerregistry import Address
from email.utils import make_msgid

# Create the base text message.
msg = EmailMessage()
msg['Subject'] = "Pourquoi pas des asperges pour ce midi ?"
msg['From'] = Address("Pepé Le Pew", "pepe", "example.com")
msg['To'] = (Address("Penelope Pussycat", "penelope", "example.com"),
             Address("Fabrette Pussycat", "fabrette", "example.com"))
msg.set_content("""\
Salut!

Cette recette [1] sera sûrement un très bon repas.

[1] http://www.yummly.com/recipe/Roasted-Asparagus-Epicurious-203718

--Pepé
""")

# Add the html version.  This converts the message into a multipart/alternative
# container, with the original text message as the first part and the new html
# message as the second part.
asparagus_cid = make_msgid()
msg.add_alternative("""\
<html>
  <head></head>
  <body>
    <p>Salut!</p>
    <p>Cette
        <a href="http://www.yummly.com/recipe/Roasted-Asparagus-Epicurious-203718">
            recette
```

(suite sur la page suivante)

(suite de la page précédente)

```
        </a> sera sûrement un très bon repas.
    </p>
    <img src="cid:{asparagus_cid}" />
  </body>
</html>
""".format(asparagus_cid=asparagus_cid[1:-1]), subtype='html')
# note that we needed to peel the <> off the msgid for use in the html.

# Now add the related image to the html part.
with open("roasted-asparagus.jpg", 'rb') as img:
    msg.get_payload()[1].add_related(img.read(), 'image', 'jpeg',
                                     cid=asparagus_cid)

# Make a local copy of what we are going to send.
with open('outgoing.msg', 'wb') as f:
    f.write(bytes(msg))

# Send the message via local SMTP server.
with smtplib.SMTP('localhost') as s:
    s.send_message(msg)
```

Si on nous avait envoyé le message de l'exemple précédent, voici la manière avec laquelle nous pourrions le traiter :

```
import os
import sys
import tempfile
import mimetypes
import webbrowser

# Import the email modules we'll need
from email import policy
from email.parser import BytesParser


def magic_html_parser(html_text, partfiles):
    """Return safety-sanitized html linked to partfiles.

    Rewrite the href="cid:...." attributes to point to the filenames in partfiles.
    Though not trivial, this should be possible using html.parser.
    """
    raise NotImplementedError("Add the magic needed")


# In a real program you'd get the filename from the arguments.
with open('outgoing.msg', 'rb') as fp:
    msg = BytesParser(policy=policy.default).parse(fp)

# Now the header items can be accessed as a dictionary, and any non-ASCII will
# be converted to unicode:
print('To:', msg['to'])
print('From:', msg['from'])
print('Subject:', msg['subject'])

# If we want to print a preview of the message content, we can extract whatever
# the least formatted payload is and print the first three lines.  Of course,
# if the message has no plain text part printing the first three lines of html
```

(suite sur la page suivante)

(suite de la page précédente)

```
# is probably useless, but this is just a conceptual example.
simplest = msg.get_body(preferencelist=('plain', 'html'))
print()
print(''.join(simplest.get_content().splitlines(keepends=True)[:3]))

ans = input("View full message?")
if ans.lower()[0] == 'n':
    sys.exit()

# We can extract the richest alternative in order to display it:
richest = msg.get_body()
partfiles = {}
if richest['content-type'].maintype == 'text':
    if richest['content-type'].subtype == 'plain':
        for line in richest.get_content().splitlines():
            print(line)
        sys.exit()
    elif richest['content-type'].subtype == 'html':
        body = richest
    else:
        print("Don't know how to display {}".format(richest.get_content_type()))
        sys.exit()
elif richest['content-type'].content_type == 'multipart/related':
    body = richest.get_body(preferencelist=('html'))
    for part in richest.iter_attachments():
        fn = part.get_filename()
        if fn:
            extension = os.path.splitext(part.get_filename())[1]
        else:
            extension = mimetypes.guess_extension(part.get_content_type())
        with tempfile.NamedTemporaryFile(suffix=extension, delete=False) as f:
            f.write(part.get_content())
            # again strip the <> to go from email form of cid to html form.
            partfiles[part['content-id'][1:-1]] = f.name
else:
    print("Don't know how to display {}".format(richest.get_content_type()))
    sys.exit()
with tempfile.NamedTemporaryFile(mode='w', delete=False) as f:
    f.write(magic_html_parser(body.get_content(), partfiles))
webbrowser.open(f.name)
os.remove(f.name)
for fn in partfiles.values():
    os.remove(fn)

# Of course, there are lots of email messages that could break this simple
# minded program, but it will handle the most common ones.
```

La sortie textuelle du code ci dessus est :

```
To: Penelope Pussycat <penelope@example.com>, Fabrette Pussycat <fabrette@example.
↪com>
From: Pepé Le Pew <pepe@example.com>
Subject: Pourquoi pas des asperges pour ce midi ?

Salut!

Cette recette [1] sera sûrement un très bon repas.
```

Notes

API héritée :

20.1.9 email.message.Message : représentation d'un message électronique à l'aide de l'API compat32

La classe *Message* est très similaire à la classe *EmailMessage*, sans les méthodes ajoutées par cette classe et avec le comportement par défaut légèrement différent pour certaines autres méthodes. Nous documentons également ici certaines méthodes qui, bien que prises en charge par la classe *EmailMessage*, ne sont pas recommandées, sauf si vous avez affaire à du code hérité.

La philosophie et la structure des deux classes sont par ailleurs les mêmes.

Ce document décrit le comportement avec les règles par défaut (pour *Message*) *Compat32*. Si vous envisagez d'utiliser d'autres règles, vous devriez plutôt utiliser la classe *EmailMessage*.

Un message électronique se compose d'*en-têtes* et d'une *charge utile*. Les en-têtes doivent être des noms et des valeurs de style **RFC 5322**, où le nom et la valeur du champ sont séparés par deux-points. Les deux-points ne font partie ni du nom du champ ni de la valeur du champ. La charge utile peut être un simple message texte, ou un objet binaire, ou une séquence structurée de sous-messages chacun avec son propre ensemble d'en-têtes et sa propre charge utile. Ce dernier type de charge utile est indiqué par le message ayant un type MIME tel que *multipart/** ou *message/rfc822*.

Le modèle conceptuel fourni par un objet *Message* est celui d'un dictionnaire ordonné d'en-têtes avec des méthodes supplémentaires pour accéder à la fois aux informations spécialisées des en-têtes, pour accéder à la charge utile, pour générer une version sérialisée du message et pour parcourir récursivement l'arbre d'objets. Notez que les en-têtes en double sont pris en charge mais que des méthodes spéciales doivent être utilisées pour y accéder.

Le pseudo-dictionnaire *Message* est indicé par les noms d'en-tête, qui doivent être des valeurs ASCII. Les valeurs du dictionnaire sont des chaînes censées contenir uniquement des caractères ASCII ; il existe une gestion spéciale pour les entrées non ASCII, mais cela ne produit pas toujours les résultats corrects. Les en-têtes sont stockés et renvoyés sous une forme respectant la casse, mais les noms de champ sont mis en correspondance sans tenir compte de la casse. Il peut également y avoir un seul en-tête d'enveloppe, également appelé en-tête *Unix-From* ou en-tête `From_`. La *charge utile* est soit une chaîne ou des octets, dans le cas d'objets message simples, soit une liste d'objets *Message*, pour les documents conteneurs MIME (par exemple *multipart/** et *message/rfc822*).

Voici les méthodes de la classe *Message* :

class email.message.**Message**(*policy=compat32*)

Si *policy* est spécifié (il doit s'agir d'une instance d'une classe *policy*), utilise les règles spécifiées pour mettre à jour et sérialiser la représentation du message. Si *policy* n'est pas défini, utilise les règles *compat32*, qui maintient la rétrocompatibilité avec la version Python 3.2 du paquet de messagerie. Pour plus d'informations, consultez la documentation *policy*.

Modifié dans la version 3.3 : l'argument nommé *policy* a été ajouté.

as_string(*unixfrom=False*, *maxheaderlen=0*, *policy=None*)

Renvoie le message entier aplati sous forme de chaîne. Lorsque l'option *unixfrom* est vraie, l'en-tête de l'enveloppe est inclus dans la chaîne renvoyée. *unixfrom* par défaut est `False`. Pour des raisons de compatibilité descendante, *maxheaderlen* est par défaut à `0`, donc si vous voulez une valeur différente, vous devez la remplacer explicitement (la valeur spécifiée pour *max_line_length* dans la politique sera ignorée par cette méthode). L'argument *policy* peut être utilisé pour remplacer les règles par défaut obtenues à partir de l'instance de message. Cela peut être utilisé pour contrôler une partie du formatage produit par la méthode, puisque la *policy* spécifiée est transmise au `Generator`.

L'aplatissement du message peut déclencher des changements dans *Message* si les valeurs par défaut doivent être renseignées pour terminer la transformation en chaîne (par exemple, les limites MIME peuvent être générées ou modifiées).

Notez que cette méthode est fournie à titre de commodité et peut ne pas toujours formater le message comme vous le souhaitez. Par exemple, par défaut, elle ne reformate pas les lignes qui commencent par `From` qui est requis par le format *mbox* Unix. Pour plus de flexibilité, instanciez une instance *Generator* et utilisez sa méthode *flatten()* directement. Par exemple :

```
from io import StringIO
from email.generator import Generator
fp = StringIO()
g = Generator(fp, mangle_from_=True, maxheaderlen=60)
g.flatten(msg)
text = fp.getvalue()
```

Si l'objet message contient des données binaires qui ne sont pas encodées selon les normes RFC, les données non conformes seront remplacées par des points de code Unicode « caractère inconnu » (voir aussi `as_bytes()` et `BytesGenerator`).

Modifié dans la version 3.4 : l'argument nommé *policy* a été ajouté.

__str__()

Equivalent to `as_string()`. Allows `str(msg)` to produce a string containing the formatted message.

as_bytes(*unixfrom=False*, *policy=None*)

Renvoie le message entier aplati en tant qu'objet bytes. Lorsque l'option *unixfrom* est vraie, l'en-tête de l'enveloppe est inclus dans la chaîne renvoyée. *unixfrom* par défaut est `False`. L'argument *policy* peut être utilisé pour remplacer les règles par défaut obtenues à partir de l'instance de message. Cela peut être utilisé pour contrôler une partie du formatage produit par la méthode, puisque la *policy* spécifiée est passée au `BytesGenerator`.

L'aplatissement du message peut déclencher des changements dans `Message` si les valeurs par défaut doivent être renseignées pour terminer la transformation en chaîne (par exemple, les limites MIME peuvent être générées ou modifiées).

Notez que cette méthode est fournie à titre de commodité et peut ne pas toujours formater le message comme vous le souhaitez. Par exemple, par défaut, elle ne reformate pas les lignes qui commencent par `From` qui est requis par le format *mbox* Unix. Pour plus de flexibilité, instanciez une instance `BytesGenerator` et utilisez sa méthode `flatten()` directement. Par exemple :

```
from io import BytesIO
from email.generator import BytesGenerator
fp = BytesIO()
g = BytesGenerator(fp, mangle_from_=True, maxheaderlen=60)
g.flatten(msg)
text = fp.getvalue()
```

Ajouté dans la version 3.4.

__bytes__()

Equivalent to `as_bytes()`. Allows `bytes(msg)` to produce a bytes object containing the formatted message.

Ajouté dans la version 3.4.

is_multipart()

Renvoie `True` si la charge utile du message est une liste d'objets sous-`Message`, sinon renvoie `False`. Lorsque `is_multipart()` renvoie `False`, la charge utile doit être un objet chaîne (qui peut être une charge utile binaire encodée CTE). (Notez que `is_multipart()` renvoyant `True` ne signifie pas nécessairement que `msg.get_content_maintype() == 'multipart'` renvoie `True`. Par exemple, `is_multipart` renvoie `True` lorsque le `Message` est de type `message/rfc822`.)

set_unixfrom(*unixfrom*)

Définit l'en-tête de l'enveloppe du message sur *unixfrom*, qui doit être une chaîne.

get_unixfrom()

Renvoie l'en-tête de l'enveloppe du message. La valeur par défaut est `None` si l'en-tête de l'enveloppe n'a jamais été défini.

attach(*payload*)

Ajoute le *payload* donné à la charge utile actuelle, qui doit être `None` ou une liste d'objets `Message` avant l'appel. Après l'appel, la charge utile sera toujours une liste d'objets `Message`. Si vous souhaitez définir la charge utile sur un objet scalaire (par exemple, une chaîne), utilisez `set_payload()` à la place.

Il s'agit d'une ancienne méthode. Dans la classe `EmailMessage` sa fonctionnalité est remplacée par `set_content()` et les méthodes associées `make` et `add`.

get_payload(*i=None*, *decode=False*)

Renvoie la charge utile actuelle, qui est une liste d'objets `Message` lorsque `is_multipart()` est `True`, ou une chaîne lorsque `is_multipart()` est `False`. Si la charge utile est une liste et que vous modifiez l'objet liste, vous modifiez la charge utile du message « sur place ».

Avec l'argument optionnel *i*, `get_payload()` renvoie le $i^{\text{ème}}$ élément de la charge utile, en partant de zéro, si `is_multipart()` vaut `True`. Une `IndexError` est levée si *i* est inférieur à 0 ou supérieur ou égal au nombre d'éléments dans la charge utile. Si la charge utile est une chaîne (c'est-à-dire que `is_multipart()` est `False`) et que *i* est donné, une `TypeError` est levée.

decode est un indicateur facultatif indiquant si la charge utile doit être décodée ou non, selon l'en-tête `Content-Transfer-Encoding`. Lorsqu'il vaut `True` et que le message n'est pas en plusieurs parties, la charge utile est décodée si la valeur de cet en-tête est `quoted-printable` ou `base64`. Si un autre encodage est utilisé, ou si l'en-tête `Content-Transfer-Encoding` est manquant, la charge utile est renvoyée telle quelle (non décodée). Dans tous les cas, la valeur renvoyée est une donnée binaire. Si le message est en plusieurs parties et que l'indicateur *decode* est `True`, alors `None` est renvoyé. Si la charge utile est en base64 et qu'elle n'a pas été parfaitement formée (remplissage manquant, caractères en dehors de l'alphabet base64), alors un défaut approprié est ajouté à la propriété de défaut du message (`InvalidBase64PaddingDefect` ou `InvalidBase64CharactersDefect`, respectivement).

Lorsque *decode* est `False` (valeur par défaut), le corps est renvoyé sous forme de chaîne sans décoder le `Content-Transfer-Encoding`. Cependant, pour un `Content-Transfer-Encoding` de 8 bits, une tentative est faite pour décoder les octets d'origine en utilisant le `charset` spécifié par l'en-tête `Content-Type`, en utilisant le gestionnaire d'erreurs `replace`. Si aucun `charset` n'est spécifié, ou si le `charset` donné n'est pas reconnu par le paquet de messagerie, le corps est décodé en utilisant le jeu de caractères ASCII par défaut.

Il s'agit d'une ancienne méthode. Dans la classe `EmailMessage` sa fonctionnalité est remplacée par `get_content()` et `iter_parts()`.

set_payload(*payload*, *charset=None*)

Définit la charge utile de l'objet message entier à *payload*. Il est de la responsabilité du client de s'assurer des invariants de la charge utile. Le *charset* facultatif définit le jeu de caractères par défaut du message ; voir `set_charset()` pour plus de détails.

Il s'agit d'une ancienne méthode. Dans la classe `EmailMessage` sa fonctionnalité est remplacée par `set_content()`.

set_charset(*charset*)

Définit le jeu de caractères de la charge utile à *charset*, qui peut être soit une instance de `Charset` (voir `email.charset`), une chaîne nommant un jeu de caractères ou `None`. S'il s'agit d'une chaîne, elle est convertie en une instance `Charset`. Si *charset* est `None`, le paramètre `charset` est supprimé de l'en-tête `Content-Type` (le message n'est pas modifié autrement). Tout le reste lève une `TypeError`.

S'il n'y a pas d'en-tête `MIME-Version` existant, un en-tête est ajouté. S'il n'y a pas d'en-tête `Content-Type`, un en-tête est ajouté avec une valeur de `text/plain`. Que l'en-tête `Content-Type` existe déjà ou non, son paramètre `charset` est défini sur *charset.output_charset*. Si *charset.input_charset* et *charset.output_charset* diffèrent, la charge utile est ré-encodée dans le *output_charset*. S'il n'y a pas d'en-tête `Content-Transfer-Encoding` existant, alors la charge utile est encodée par transfert, si nécessaire, en utilisant le `Charset` spécifié et un en-tête avec cette valeur est ajouté. Si un en-tête `Content-Transfer-Encoding` existe déjà, la charge utile est supposée être déjà correctement encodée à l'aide de cet en-tête `Content-Transfer-Encoding` et n'est pas modifiée.

Il s'agit d'une ancienne méthode. Dans la classe `EmailMessage` sa fonctionnalité est remplacée par le paramètre *charset* de la méthode `email.emailmessage.EmailMessage.set_content()`.

get_charset()

Renvoie l'instance `Charset` associée à la charge utile du message.

Il s'agit d'une ancienne méthode. Dans la classe `EmailMessage`, elle renvoie toujours `None`.

Les méthodes suivantes implémentent une interface de type correspondance pour accéder aux en-têtes **RFC 2822** du message. Notez qu'il existe des différences sémantiques entre ces méthodes et une interface de correspondance normale (c'est-à-dire un dictionnaire). Par exemple, dans un dictionnaire, il n'y a pas de clés en double, mais ici, il peut y avoir des en-têtes de message en double. De plus, dans les dictionnaires, il n'y a pas d'ordre garanti pour les clés renvoyées par `keys()`, mais dans un objet `Message`, les en-têtes sont toujours renvoyés dans l'ordre dans lequel ils sont apparus dans le message d'origine, ou ont été ajoutés au message plus tard. Tout en-tête supprimé puis rajouté est toujours ajouté à la fin de la liste des en-têtes.

Ces différences sémantiques sont intentionnelles et visent une commodité maximale.

Notez que dans tous les cas, tout en-tête d'enveloppe présent dans le message n'est pas inclus dans l'interface de correspondance.

Dans un modèle généré à partir d'octets, toutes les valeurs d'en-tête qui (en violation des RFC) contiennent des octets non-ASCII sont, lorsqu'elles sont récupérées via cette interface, représentées comme des objets `Header` avec un *charset* à `unknown-8bit`.

__len__()

Renvoie le nombre total d'en-têtes, y compris les doublons.

__contains__(*name*)

Renvoie `True` si l'objet message a un champ nommé *name*. La correspondance est insensible à la casse et *name* ne doit pas inclure les deux-points à la fin. Utilisé pour l'opérateur `in`, par exemple :

```
if 'message-id' in myMessage:
   print('Message-ID:', myMessage['message-id'])
```

__getitem__(*name*)

Renvoie la valeur du champ d'en-tête désigné. *name* ne doit pas inclure le séparateur de champ deux-points. Si l'en-tête est manquant, `None` est renvoyé ; `KeyError` n'est jamais levée.

Notez que si le champ désigné apparaît plus d'une fois dans les en-têtes du message, la valeur exacte renvoyée n'est pas définie. Utilisez la méthode `get_all()` pour obtenir les valeurs de tous les en-têtes existants.

__setitem__(*name*, *val*)

Ajoute un en-tête au message avec le nom de champ *name* et la valeur *val*. Le champ est ajouté à la fin des champs existants du message.

Notez que cela n'écrase *pas* ou ne supprime aucun en-tête existant portant le même nom. Si vous voulez vous assurer que le nouvel en-tête est le seul présent dans le message avec le nom de champ *name*, supprimez d'abord le champ, par exemple :

```
del msg['subject']
msg['subject'] = 'Python roolz!'
```

__delitem__(*name*)

Supprime toutes les occurrences du champ portant le nom *name* des en-têtes du message. Aucune exception n'est levée si le champ désigné n'est pas présent dans les en-têtes.

keys()

Renvoie une liste de tous les noms de champs d'en-tête du message.

values()

Renvoie une liste de toutes les valeurs de champ du message.

items()

Renvoie une liste de couples contenant tous les en-têtes et valeurs de champs associées du message.

get(*name*, *failobj=None*)

Return the value of the named header field. This is identical to `__getitem__()` except that optional *failobj* is returned if the named header is missing (defaults to `None`).

Voici quelques méthodes supplémentaires utiles :

get_all(*name*, *failobj=None*)

Renvoie la liste de toutes les valeurs du champ nommé *name*. S'il n'y a pas d'en-têtes de ce nom dans le message, *failobj* est renvoyé (la valeur par défaut est `None`).

add_header(*_name*, *_value*, ***_params*)

Définition de paramètres d'en-têtes étendus. Cette méthode est similaire à *`__setitem__()`* sauf que des paramètres d'en-tête supplémentaires peuvent être fournis en tant qu'arguments nommés. *_name* est le champ d'en-tête à ajouter et *_value* est la valeur *primaire* de l'en-tête.

Pour chaque élément du dictionnaire d'arguments nommés *_params*, la clé est prise comme nom de paramètre, avec des traits de soulignement convertis en tirets (puisque les tirets sont illégaux dans les identifiants Python). Normalement, le paramètre est ajouté en tant que `key="value"` sauf si la valeur est `None`, auquel cas seule la clé est ajoutée. Si la valeur contient des caractères non-ASCII, elle peut être spécifiée sous la forme d'un triplet au format `(CHARSET, LANGUAGE, VALUE)`, où `CHARSET` est une chaîne nommant le jeu de caractères à utiliser pour coder la valeur, `LANGUAGE` peut généralement être défini sur `None` ou la chaîne vide (voir la **RFC 2231** pour d'autres possibilités), et `VALUE` est la valeur de chaîne contenant des points de code non-ASCII. Si un triplet n'est pas passé et que la valeur contient des caractères non-ASCII, elle est automatiquement encodée au format **RFC 2231** en utilisant un `CHARSET` à `utf-8` et un `LANGUAGE` à `None`.

Voici un exemple :

```
msg.add_header('Content-Disposition', 'attachment', filename='bud.gif')
```

Cela ajoute un en-tête qui ressemble à :

```
Content-Disposition: attachment; filename="bud.gif"
```

Un exemple avec des caractères non-ASCII :

```
msg.add_header('Content-Disposition', 'attachment',
               filename=('iso-8859-1', '', 'Fußballer.ppt'))
```

qui produit :

```
Content-Disposition: attachment; filename*="iso-8859-1''Fu%DFballer.ppt"
```

replace_header(*_name*, *_value*)

Remplace un en-tête. Remplace le premier en-tête trouvé dans le message qui correspond à *_name*, en conservant l'ordre des en-têtes et la casse des noms de champs. Si aucun en-tête correspondant n'a été trouvé, une *`KeyError`* est levée.

get_content_type()

Renvoie le type de contenu du message. La chaîne renvoyée est convertie en minuscules sous la forme *`maintype/subtype`*. S'il n'y avait pas d'en-tête *`Content-Type`* dans le message, le type par défaut tel qu'indiqué par *`get_default_type()`* est renvoyé. Puisque selon la **RFC 2045**, les messages ont toujours un type par défaut, *`get_content_type()`* renvoie toujours une valeur.

La **RFC 2045** définit le type par défaut d'un message comme étant *`text/plain`* sauf s'il apparaît dans un conteneur *`multipart/digest`*, auquel cas ce serait *`message/rfc822`*. Si l'en-tête *`Content-Type`* a une spécification de type invalide, la **RFC 2045** exige que le type par défaut soit *`text/plain`*.

get_content_maintype()

Renvoie le type de contenu principal du message. C'est la partie *`maintype`* de la chaîne renvoyée par *`get_content_type()`*.

get_content_subtype()

Renvoie le type de sous-contenu du message. C'est la partie *subtype* de la chaîne renvoyée par *get_content_type()*.

get_default_type()

Renvoie le type de contenu par défaut. La plupart des messages ont un type de contenu par défaut de *text/plain*, à l'exception des messages qui sont des sous-parties des conteneurs *multipart/digest*. Ces sous-parties ont un type de contenu par défaut de *message/rfc822*.

set_default_type(*ctype*)

Définit le type de contenu par défaut. *ctype* doit être *text/plain* ou *message/rfc822*, bien que cela ne soit pas appliqué. Le type de contenu par défaut n'est pas stocké dans l'en-tête *Content-Type*.

get_params(*failobj=None*, *header='content-type'*, *unquote=True*)

Renvoie les paramètres *Content-Type* du message, sous forme de liste. Les éléments de la liste renvoyée sont des couples clé-valeur, ayant été séparés sur le signe `'='`. Le côté gauche du `'='` est la clé, tandis que le côté droit est la valeur. S'il n'y a pas de signe `'='` dans le paramètre, la valeur est la chaîne vide, sinon la valeur est telle que décrite dans *get_param()* et n'est pas entre guillemets si l'option *unquote* est `True` (la valeur par défaut).

failobj (facultatif) est l'objet à renvoyer s'il n'y a pas d'en-tête *Content-Type*. *header* (facultatif) est l'en-tête à rechercher au lieu de *Content-Type*.

Il s'agit d'une ancienne méthode. Dans la classe `EmailMessage`, sa fonctionnalité est remplacée par la propriété *params* des objets d'en-tête individuels renvoyés par les méthodes d'accès aux en-têtes.

get_param(*param*, *failobj=None*, *header='content-type'*, *unquote=True*)

Renvoie la valeur du paramètre *param* de l'en-tête *Content-Type* sous forme de chaîne. Si le message n'a pas d'en-tête *Content-Type* ou s'il n'y a pas un tel paramètre, alors *failobj* est renvoyé (par défaut à `None`).

header s'il est fourni, spécifie l'en-tête de message à utiliser à la place de *Content-Type*.

Les clés en paramètre sont toujours comparées sans tenir compte de la casse. La valeur de retour peut être soit une chaîne, soit un triplet si le paramètre a été encodé selon la **RFC 2231**. Lorsqu'il s'agit d'un triplet, les éléments de la valeur sont de la forme `(CHARSET, LANGUAGE, VALUE)`. Notez que `CHARSET` et `LANGUAGE` peuvent être `None`, vous devriez alors considérer que `VALUE` est encodé dans le jeu de caractères `us-ascii`. Vous pouvez généralement ignorer `LANGUAGE`.

Si votre application ne se soucie pas de savoir si le paramètre a été encodé conformément à la **RFC 2231**, vous pouvez réduire la valeur du paramètre en appelant *email.utils.collapse_rfc2231_value()*, en transmettant la valeur de retour de *get_param()*. Cela renverra une chaîne Unicode convenablement décodée lorsque la valeur est un *n*-uplet, ou la chaîne d'origine sans guillemets si ce n'est pas le cas. Par exemple :

```
rawparam = msg.get_param('foo')
param = email.utils.collapse_rfc2231_value(rawparam)
```

Dans tous les cas, la valeur du paramètre (soit la chaîne renvoyée, soit l'élément `VALUE` dans le triplet) est toujours sans guillemets, sauf si *unquote* est défini sur `False`.

Il s'agit d'une ancienne méthode. Dans la classe `EmailMessage`, sa fonctionnalité est remplacée par la propriété *params* des objets d'en-tête individuels renvoyés par les méthodes d'accès aux en-têtes.

set_param(*param*, *value*, *header='Content-Type'*, *requote=True*, *charset=None*, *language=''*, *replace=False*)

Définit un paramètre dans l'en-tête *Content-Type*. Si le paramètre existe déjà dans l'en-tête, sa valeur est remplacée par *value*. Si l'en-tête *Content-Type* n'a pas encore été défini pour ce message, il est défini à *text/plain* et la nouvelle valeur du paramètre est ajoutée conformément à la **RFC 2045**.

L'option *header* spécifie un en-tête alternatif à *Content-Type*, et tous les paramètres seront entre guillemets si nécessaire sauf si l'option *requote* est `False` (la valeur par défaut est `True`).

Si *charset* est spécifié, le paramètre est encodé selon la **RFC 2231**. *language* (facultatif) spécifie la langue RFC 2231, par défaut c'est la chaîne vide. *charset* et *language* doivent être des chaînes.

Si *replace* est `False` (valeur par défaut), l'en-tête est déplacé à la fin de la liste des en-têtes. Si *replace* est `True`, l'en-tête est mis à jour « sur place ».

Modifié dans la version 3.4 : l'argument nommé `replace` a été ajouté.

del_param(*param*, *header='content-type'*, *requote=True*)

Supprime complètement le paramètre donné de l'en-tête `Content-Type`. L'en-tête est réécrit « sur place » sans le paramètre ou sa valeur. Toutes les valeurs seront entre guillemets si nécessaire sauf si *requote* est `False` (la valeur par défaut est `True`). L'option *header* spécifie une alternative à `Content-Type`.

set_type(*type*, *header='Content-Type'*, *requote=True*)

Définit le type principal et le sous-type de l'en-tête `Content-Type`. *type* doit être une chaîne sous la forme `maintype/subtype`, sinon une `ValueError` est levée.

Cette méthode remplace l'en-tête `Content-Type`, en gardant tous les paramètres en place. Si *requote* est `False`, cela laisse les guillemets de l'en-tête existant tels quels, sinon les paramètres sont mis entre guillemets (par défaut).

Un en-tête alternatif peut être spécifié dans l'argument *header*. Lorsque l'en-tête `Content-Type` est défini, un en-tête `MIME-Version` est également ajouté.

Il s'agit d'une ancienne méthode. Dans la classe `EmailMessage` sa fonctionnalité est remplacée par les méthodes `make_` et `add_`.

get_filename(*failobj=None*)

Renvoie la valeur du paramètre `filename` de l'en-tête `Content-Disposition` du message. Si l'en-tête n'a pas de paramètre `filename`, cette méthode revient à rechercher le paramètre `name` dans l'en-tête `Content-Type`. Si aucun n'est trouvé, ou si l'en-tête est manquant, alors *failobj* est renvoyé. La chaîne renvoyée est toujours sans guillemets comme indiqué par `email.utils.unquote()`.

get_boundary(*failobj=None*)

Renvoie la valeur du paramètre `boundary` de l'en-tête `Content-Type` du message, ou *failobj* si l'en-tête est manquant ou n'a pas de paramètre `boundary`. La chaîne renvoyée est toujours sans guillemets comme indiqué par `email.utils.unquote()`.

set_boundary(*boundary*)

Définit le paramètre `boundary` de l'en-tête `Content-Type` sur *boundary*. `set_boundary()` met toujours *boundary* entre guillemets si nécessaire. Une `HeaderParseError` est levée si l'objet message n'a pas d'en-tête `Content-Type`.

Notez que l'utilisation de cette méthode est légèrement différente de la suppression de l'ancien en-tête `Content-Type` et de l'ajout d'un nouveau avec la nouvelle limite via `add_header()`, car `set_boundary()` préserve l'ordre des en-têtes `Content-Type` dans la liste des en-têtes. Cependant, elle ne conserve *pas* les lignes de continuation qui auraient pu être présentes dans l'en-tête original `Content-Type`.

get_content_charset(*failobj=None*)

Renvoie le paramètre `charset` de l'en-tête `Content-Type`, mis en minuscules. S'il n'y a pas d'en-tête `Content-Type`, ou si cet en-tête n'a pas de paramètre `charset`, *failobj* est renvoyé.

Notez que cette méthode diffère de `get_charset()` qui renvoie l'instance `Charset` avec l'encodage par défaut du corps du message.

get_charsets(*failobj=None*)

Renvoie une liste contenant les noms des jeux de caractères dans le message. Si le message est un `multipart`, alors la liste contient un élément pour chaque sous-partie dans la charge utile, sinon, c'est une liste de longueur 1.

Chaque élément de la liste est une chaîne qui est la valeur du paramètre `charset` dans l'en-tête `Content-Type` pour la sous-partie représentée. Cependant, si la sous-partie n'a pas d'en-tête `Content-Type`, pas de paramètre `charset`, ou n'est pas du type MIME principal `text`, alors cet élément dans la liste renvoyée est *failobj*.

get_content_disposition()

Renvoie la valeur en minuscules (sans paramètres) de l'en-tête *Content-Disposition* du message s'il en a un, ou `None`. Les valeurs possibles pour cette méthode sont *inline*, *attachment* ou `None` si le message suit la **RFC 2183**.

Ajouté dans la version 3.5.

walk()

La méthode *walk()* est un générateur polyvalent qui peut être utilisé pour itérer sur toutes les parties et sous-parties d'une arborescence d'objets de message, dans l'ordre de parcours en profondeur d'abord. Vous utiliserez généralement *walk()* comme itérateur dans une boucle `for` ; chaque itération renvoie la sous-partie suivante.

Voici un exemple qui imprime le type MIME de chaque partie d'une structure de message en plusieurs parties :

```
>>> for part in msg.walk():
...     print(part.get_content_type())
multipart/report
text/plain
message/delivery-status
text/plain
text/plain
message/rfc822
text/plain
```

`walk` itère sur les sous-parties de toute partie où *is_multipart()* renvoie `True`, même si `msg.get_content_maintype() == 'multipart'` peut renvoyer `False`. Nous pouvons le voir dans notre exemple en utilisant la fonction d'aide au débogage `_structure` :

```
>>> for part in msg.walk():
...     print(part.get_content_maintype() == 'multipart',
...           part.is_multipart())
True True
False False
False True
False False
False False
False True
False False
>>> _structure(msg)
multipart/report
    text/plain
    message/delivery-status
        text/plain
        text/plain
    message/rfc822
        text/plain
```

Ici, les parties `message` ne sont pas des `multiparts`, mais elles contiennent des sous-parties. `is_multipart()` renvoie `True` et `walk` descend dans les sous-parties.

Les objets *Message* peuvent aussi éventuellement contenir deux attributs d'instance, qui peuvent être utilisés lors de la génération du texte brut d'un message MIME.

preamble

Le format d'un document MIME permet d'insérer du texte entre la ligne vide suivant les en-têtes et la première chaîne de délimitation en plusieurs parties. Normalement, ce texte n'est jamais visible dans un lecteur de courrier compatible MIME car il ne fait pas partie du cadre MIME standard. Toutefois, lors de l'affichage du texte brut du message ou lors de l'affichage du message dans un lecteur non compatible MIME, ce texte peut devenir visible.

L'attribut *preamble* contient ce texte hors-cadre de tête pour les documents MIME. Lorsque `Parser` découvre du texte après les en-têtes mais avant la première chaîne de délimitation, il attribue ce texte à l'attribut *preamble* du message. Lorsque `Generator` écrit la représentation en texte brut d'un message MIME, et qu'il trouve que le message a un attribut *preamble*, il écrit ce texte dans la zone entre les en-têtes et la première frontière. Voir `email.parser` et `email.generator` pour plus de détails.

Notez que si l'objet message n'a pas de préambule, l'attribut *preamble* est `None`.

epilogue

L'attribut *epilogue* agit de la même manière que l'attribut *preamble*, sauf qu'il contient du texte qui apparaît entre la dernière limite et la fin du message.

Vous n'avez pas besoin de définir l'épilogue sur la chaîne vide pour que `Generator` imprime une nouvelle ligne à la fin du fichier.

defects

L'attribut *defects* contient une liste de tous les problèmes rencontrés lors de l'analyse de ce message. Voir `email.errors` pour une description détaillée des défauts d'analyse possibles.

20.1.10 `email.mime` : Creating email and MIME objects from scratch

Code source : Lib/email/mime/

Ce module fait partie de l'ancienne API de messagerie (`Compat32`). Sa fonctionnalité est partiellement remplacée par le `contentmanager` dans la nouvelle API mais, dans certaines applications, ces classes peuvent toujours être utiles, même dans du code pas si ancien.

Habituellement, vous obtenez une structure d'objet message en passant un fichier ou du texte à un analyseur, qui analyse le texte et renvoie l'objet message racine. Cependant, vous pouvez également créer une structure de message complète à partir de zéro, ou même des objets individuels `Message` à la main. En fait, vous pouvez également prendre une structure existante et ajouter de nouveaux objets `Message`, les déplacer, etc. Cela crée une interface très pratique pour découper et manipuler chaque morceau de messages MIME.

Vous pouvez créer une nouvelle structure d'objet en créant des instances `Message`, en ajoutant manuellement les pièces jointes et tous les en-têtes appropriés. Pour les messages MIME cependant, le paquet `email` fournit quelques sous-classes pratiques pour faciliter les choses.

Voici les classes :

class email.mime.base.**MIMEBase**(*_maintype*, *_subtype*, *, *policy=compat32*, ***_params*)

Module : `email.mime.base`

C'est la classe mère pour toutes les sous-classes spécifiques à MIME de `Message`. Normalement, vous ne créerez pas d'instances spécifiques de `MIMEBase`, bien que cela soit possible. `MIMEBase` est fourni principalement comme une classe mère pratique pour des sous-classes plus spécifiques compatibles MIME.

_maintype est le type majeur du `Content-Type` (par exemple `text` ou `image`) et *_subtype* est le type mineur du `Content-Type` (par exemple, `plain` ou `gif`). *_params* est un paramètre dictionnaire clé-valeur et est transmis directement à `Message.add_header`.

Si *policy* est spécifiée, (par défaut la politique `compat32`) elle est passée à `Message`.

La classe `MIMEBase` ajoute toujours un en-tête `Content-Type` (basé sur *_maintype*, *_subtype* et *_params*) et un en-tête `MIME-Version` (toujours défini à `1.0`).

Modifié dans la version 3.6 : ajout du paramètre nommé *policy*.

class email.mime.nonmultipart.**MIMENonMultipart**

Module : `email.mime.nonmultipart`

Sous-classe de `MIMEBase`, c'est une classe mère intermédiaire pour les messages MIME qui ne sont pas `multipart`. Le but principal de cette classe est d'empêcher l'utilisation de la méthode `attach()`, qui n'a de sens que pour les messages `multipart`. Si `attach()` est appelée, une exception `MultipartConversionError` est levée.

class email.mime.multipart.**MIMEMultipart**(*_subtype='mixed'*, *boundary=None*, *_subparts=None*, *, *policy=compat32*, ***_params*)

Module : email.mime.multipart

Sous-classe de MIMEBase, c'est une classe mère intermédiaire pour les messages MIME qui sont multipart. *_subtype* (facultatif) est par défaut mixed, mais peut être utilisé pour spécifier le sous-type du message. Un en-tête Content-Type de multipart/_subtype sera ajouté à l'objet message. Un en-tête MIME-Version sera également ajouté.

boundary (facultatif) est la chaîne de délimitation en plusieurs parties. Si elle vaut None (la valeur par défaut), la délimitation est calculée au besoin (par exemple, lorsque le message est sérialisé).

_subparts est une séquence de sous-parties initiales pour la charge utile. Il doit être possible de convertir cette séquence en une liste. Vous pouvez toujours joindre de nouvelles sous-parties au message en utilisant la méthode Message.attach.

L'argument facultatif *policy* est par défaut compat32.

Des paramètres supplémentaires pour l'en-tête Content-Type sont extraits des arguments nommés ou passés à l'argument *_params*, qui est un dictionnaire de mots-clés.

Modifié dans la version 3.6 : ajout du paramètre nommé *policy*.

class email.mime.application.**MIMEApplication**(*_data*, *_subtype='octet-stream'*, *_encoder=email.encoders.encode_base64*, *, *policy=compat32*, ***_params*)

Module : email.mime.application

Sous-classe de MIMENonMultipart, la classe MIMEApplication est utilisée pour représenter les objets de message MIME de type principal application. *_data* contient les octets pour les données d'application brutes. L'option *_subtype* spécifie le sous-type MIME et la valeur par défaut est octet-stream.

_encoder (facultatif) est un appelable (c'est-à-dire une fonction) qui effectue le codage réel des données pour le transport. Cet appelable prend un argument, qui est l'instance MIMEApplication. Il doit utiliser get_payload() et set_payload() pour modifier la charge utile sous forme codée. Il doit également ajouter n'importe quel Content-Transfer-Encoding ou d'autres en-têtes à l'objet message si nécessaire. L'encodage par défaut est base64. Voir le module email.encoders pour une liste des encodeurs intégrés.

L'argument facultatif *policy* est par défaut compat32.

Les *_params* sont transmis directement au constructeur de la classe mère.

Modifié dans la version 3.6 : ajout du paramètre nommé *policy*.

class email.mime.audio.**MIMEAudio**(*_audiodata*, *_subtype=None*, *_encoder=email.encoders.encode_base64*, *, *policy=compat32*, ***_params*)

Module : email.mime.audio

Sous-classe de MIMENonMultipart, la classe MIMEAudio est utilisée pour créer des objets de message MIME de type majeur audio. *_audiodata* contient les octets pour les données audio brutes. Si ces données peuvent être décodées au format *au*, *wav*, *aiff* ou *aifc*, alors le sous-type est automatiquement inclus dans l'en-tête Content-Type. Sinon, vous pouvez spécifier explicitement le sous-type audio via l'argument *_subtype*. Si le type mineur n'a pas pu être deviné et que *_subtype* n'a pas été donné, alors une TypeError est levée.

_encoder (facultatif) est un appelable (c'est-à-dire une fonction) qui effectue le codage réel des données audio pour le transport. Cet appelable prend un argument, qui est l'instance MIMEAudio. Il doit utiliser get_payload() et set_payload() pour modifier la charge utile sous forme codée. Il doit également ajouter n'importe quel Content-Transfer-Encoding ou d'autres en-têtes à l'objet message si nécessaire. L'encodage par défaut est base64. Voir le module email.encoders pour une liste des encodeurs intégrés.

L'argument facultatif *policy* est par défaut compat32.

Les *_params* sont transmis directement au constructeur de la classe mère.

Modifié dans la version 3.6 : ajout du paramètre nommé *policy*.

class email.mime.image.**MIMEImage**(*_imagedata*, *_subtype=None*, *_encoder=email.encoders.encode_base64*, *, *policy=compat32*, ***_params*)

Module : email.mime.image

Sous-classe de MIMENonMultipart, la classe MIMEImage est utilisée pour créer des objets de message MIME de type principal image. *_imagedata* contient les octets pour les données d'image brutes. Si ce type de données peut être détecté (*jpeg*, *png*, *gif*, *tiff*, *rgb*, *pbm*, *pgm*, *ppm*, *rast*, *xbm*, *bmp*, *webp* et *exr* sont essayés), alors le sous-type est automatiquement inclus dans l'en-tête Content-Type. Sinon, vous pouvez spécifier explicitement le sous-type d'image via l'argument *_subtype*. Si le type mineur n'a pas pu être deviné et que *_subtype* n'a pas été donné, alors une TypeError est levée.

_encoder (facultatif) est un appelable (c'est-à-dire une fonction) qui effectue le codage réel des données d'image pour le transport. Cet appelable prend un argument, qui est l'instance MIMEImage. Il doit utiliser get_payload() et set_payload() pour modifier la charge utile sous forme codée. Il doit également ajouter n'importe quel Content-Transfer-Encoding ou d'autres en-têtes à l'objet message si nécessaire. L'encodage par défaut est base64. Voir le module email.encoders pour une liste des encodeurs intégrés.

L'argument facultatif *policy* est par défaut compat32.

Les *_params* sont transmis directement au constructeur MIMEBase.

Modifié dans la version 3.6 : ajout du paramètre nommé *policy*.

class email.mime.message.**MIMEMessage**(*_msg*, *_subtype='rfc822'*, *, *policy=compat32*)

Module : email.mime.message

Sous-classe de MIMENonMultipart, la classe MIMEMessage est utilisée pour créer des objets MIME de type principal message. *_msg* est utilisé comme charge utile et doit être une instance de la classe Message (ou une sous-classe de celle-ci), sinon une TypeError est levée.

L'option *_subtype* définit le sous-type du message ; par défaut c'est rfc822.

L'argument facultatif *policy* est par défaut compat32.

Modifié dans la version 3.6 : ajout du paramètre nommé *policy*.

class email.mime.text.**MIMEText**(*_text*, *_subtype='plain'*, *_charset=None*, *, *policy=compat32*)

Module : email.mime.text

Sous-classe de MIMENonMultipart, la classe MIMEText est utilisée pour créer des objets MIME de type principal text. *_text* est la chaîne de la charge utile. *_subtype* est le type mineur et par défaut est plain. *_charset* est le jeu de caractères du texte et est passé en argument au constructeur MIMENonMultipart ; sa valeur par défaut est us-ascii si la chaîne ne contient que des points de code ascii, et utf-8 sinon. Le paramètre *_charset* accepte soit une chaîne soit une instance Charset.

À moins que l'argument *_charset* ne soit explicitement défini sur None, l'objet *MIMEText* créé possède à la fois un en-tête Content-Type avec un paramètre charset et un en-tête Content-Transfer-Encoding. Cela signifie qu'un appel ultérieur à set_payload n'entraîne pas l'encodage de la charge utile, même si un jeu de caractères est passé dans la commande set_payload. Vous pouvez « réinitialiser » ce comportement en supprimant l'en-tête Content-Transfer-Encoding, après quoi un appel set_payload encodera automatiquement la nouvelle charge utile (et ajoutera un nouvel en-tête Content-Transfer-Encoding).

L'argument facultatif *policy* est par défaut compat32.

Modifié dans la version 3.5 : *_charset* accepte également les instances Charset.

Modifié dans la version 3.6 : ajout du paramètre nommé *policy*.

20.1.11 email.header : Internationalized headers

Code source : Lib/email/headers.py

Ce module fait partie de l'ancienne API de messagerie (Compat32). Dans l'API actuelle, l'encodage et le décodage des en-têtes sont gérés de manière transparente par l'API de type dictionnaire de la classe EmailMessage. En plus des

utilisations dans de l'ancien code, ce module peut être utile dans les applications qui doivent contrôler complètement les jeux de caractères utilisés lors de l'encodage des en-têtes.

Le texte restant de cette section est la documentation originale de ce module.

La **RFC 2822** est la norme de base qui décrit le format des messages électroniques. Elle dérive de l'ancienne norme **RFC 822** qui s'est généralisée à une époque où la plupart des e-mails étaient composés uniquement de caractères ASCII. La **RFC 2822** est une spécification écrite en supposant que le courrier électronique ne contient que des caractères ASCII 7 bits.

Bien sûr, à mesure que le courrier électronique a été déployé dans le monde, il s'est internationalisé, de sorte que des jeux de caractères spécifiques à une langue peuvent désormais être utilisés dans les messages électroniques. La norme de base exige toujours que les messages électroniques soient transférés en utilisant uniquement des caractères ASCII 7 bits, de sorte qu'un grand nombre de RFC décrivent comment encoder les e-mails contenant des caractères non ASCII dans le format conforme à la **RFC 2822**. Ces RFC incluent la **RFC 2045**, la **RFC 2046**, la **RFC 2047** et la **RFC 2231**. Le paquet `email` gère ces normes dans ses modules `email.header` et `email.charset`.

Si vous souhaitez inclure des caractères non-ASCII dans les en-têtes de vos e-mails, par exemple dans les champs `Subject` ou `To`, vous devez utiliser la classe `Header` et affecter le champ dans le `Message` à une instance de `Header` au lieu d'utiliser une chaîne pour la valeur de l'en-tête. Importez la classe `Header` du module `email.header`. Par exemple :

```
>>> from email.message import Message
>>> from email.header import Header
>>> msg = Message()
>>> h = Header('p\xf6stal', 'iso-8859-1')
>>> msg['Subject'] = h
>>> msg.as_string()
'Subject: =?iso-8859-1?q?p=F6stal?=\n\n'
```

Remarquez ici comment nous voulions que le champ `Subject` contienne un caractère non-ASCII : nous avons créé une instance `Header` et transmis le jeu de caractères dans lequel la chaîne d'octets était encodée. Lorsque l'instance suivante `Message` a été aplatie, le champ `Subject` était correctement encodé selon la **RFC 2047**. Les lecteurs de courrier compatibles MIME afficheraient cet en-tête en utilisant le caractère ISO-8859-1 intégré.

Voici la description de la classe `Header` :

class email.header.**Header**(*s=None*, *charset=None*, *maxlinelen=None*, *header_name=None*, *continuation_ws=' '*, *errors='strict'*)

Crée un en-tête compatible MIME pouvant contenir des chaînes dans différents jeux de caractères.

s (facultatif) est la valeur d'en-tête initiale. S'il vaut `None` (valeur par défaut), la valeur d'en-tête initiale n'est pas définie. Vous pouvez ensuite ajouter des choses à l'en-tête avec des appels de méthode `append()`. *s* peut être une instance de `bytes` ou `str`, mais consultez la documentation `append()` pour la sémantique.

L'option *charset* sert à deux fins : elle a la même signification que l'argument *charset* de la méthode `append()`. Elle définit également le jeu de caractères par défaut pour tous les appels suivants `append()` qui omettent l'argument *charset*. Si *charset* n'est pas fourni dans le constructeur (par défaut), le jeu de caractères `us-ascii` est utilisé à la fois comme jeu de caractères initial de *s* et comme jeu par défaut pour les appels suivants à `append()`.

The maximum line length can be specified explicitly via *maxlinelen*. For splitting the first line to a shorter value (to account for the field header which isn't included in *s*, e.g. `Subject`) pass in the name of the field in *header_name*. The default *maxlinelen* is 78, and the default value for *header_name* is `None`, meaning it is not taken into account for the first line of a long, split header.

continuation_ws (facultatif) doit être conforme aux caractères de reformatage **RFC 2822** ; c'est généralement soit une espace, soit un caractère de tabulation fixe. Ce caractère est ajouté aux lignes de continuation. *continuation_ws* utilise par défaut une seule espace.

errors (facultatif) est transmis directement à la méthode `append()`.

append(*s*, *charset=None*, *errors='strict'*)

Ajoute la chaîne *s* à l'en-tête MIME.

Le *charset* facultatif, s'il est fourni, doit être une instance de `Charset` (voir `email.charset`) ou le nom d'un jeu de caractères, qui est converti en une instance `Charset`. Une valeur `None` (la valeur par défaut) signifie que le *charset* donné dans le constructeur est utilisé.

s peut être une instance de `bytes` ou `str`. S'il s'agit d'une instance de `bytes`, alors *charset* est l'encodage de cette chaîne d'octets et une `UnicodeError` est levée si la chaîne ne peut pas être décodée avec ce jeu de caractères.

Si *s* est une instance de `str`, alors *charset* est une indication spécifiant le jeu de caractères des caractères de la chaîne.

Dans les deux cas, lors de la production d'un en-tête conforme à la **RFC 2822** à l'aide des règles **RFC 2047**, la chaîne est encodée à l'aide du codec de sortie du jeu de caractères. Si la chaîne ne peut pas être encodée à l'aide du codec de sortie, une erreur *UnicodeError* est levée.

L'option *errors* est transmise comme argument d'erreurs à l'appel de décodage si *s* est une chaîne d'octets.

encode(*splitchars=';, \t'*, *maxlinelen=None*, *linesep='\n'*)

Encode un en-tête de message dans un format conforme à la RFC, en reformatant éventuellement de longues lignes et en encapsulant des parties non ASCII dans des encodages base64 ou dits *quoted-printable* (c.-à-d. que, par exemple, les caractères non ASCII sont représentés par un signe égal, suivi de son numéro, exprimé en hexadécimal).

splitchars (facultatif) est une chaîne contenant des caractères auxquels l'algorithme de fractionnement doit donner un poids supplémentaire lors du reformatage normal de l'en-tête. Il s'agit d'une prise en charge très approximative des « ruptures syntaxiques de niveau supérieur » de la **RFC 2822** : les points de séparation précédés d'un caractère de séparation sont préférés lors de la séparation des lignes, les caractères étant préférés dans l'ordre dans lequel ils apparaissent dans la chaîne. Une espace et une tabulation peuvent être incluses dans la chaîne pour indiquer si la préférence doit être donnée à l'une plutôt qu'à l'autre comme point de partage lorsque d'autres caractères fractionnés n'apparaissent pas dans la ligne fractionnée. *splitchars* n'affecte pas les lignes encodées selon la **RFC 2047**.

maxlinelen, s'il est fourni, remplace la valeur de l'instance pour la longueur de ligne maximale.

linesep spécifie les caractères utilisés pour séparer les lignes de l'en-tête plié. Il prend par défaut la valeur la plus utile pour le code d'application Python (`\n`), mais `\r\n` peut être spécifié afin de produire des en-têtes avec des séparateurs de ligne conformes à la RFC.

Modifié dans la version 3.2 : ajout de l'argument *linesep*.

La classe `Header` fournit également un certain nombre de méthodes pour prendre en charge les opérateurs standard et les fonctions intégrées.

__str__()

Renvoie une approximation de `Header` sous forme de chaîne, en utilisant une longueur de ligne illimitée. Toutes les parties sont converties en Unicode en utilisant l'encodage spécifié et réunies de manière appropriée. Tous les morceaux avec un *charset* à `'unknown-8bit'` sont décodés en ASCII en utilisant le gestionnaire d'erreurs `'replace'`.

Modifié dans la version 3.2 : ajout de la gestion du jeu de caractères `'unknown-8bit'`.

__eq__(*other*)

Cette méthode vous permet de tester l'égalité de deux instances de `Header`.

__ne__(*other*)

Cette méthode vous permet de tester l'inégalité de deux instances de `Header`.

Le module `email.header` fournit également les fonctions pratiques suivantes.

email.header.**decode_header**(*header*)

Décode une valeur d'en-tête de message sans convertir le jeu de caractères. La valeur de l'en-tête est dans *header*.

Cette fonction renvoie une liste de paires `(decoded_string, charset)` contenant chacune des parties décodées de l'en-tête. *charset* est `None` pour les parties non encodées de l'en-tête, sinon une chaîne en minuscules contenant le nom du jeu de caractères spécifié dans la chaîne encodée.

Voici un exemple :

```
>>> from email.header import decode_header
>>> decode_header('=?iso-8859-1?q?p=F6stal?=')
[(b'p\xf6stal', 'iso-8859-1')]
```

email.header.**make_header**(*decoded_seq*, *maxlinelen=None*, *header_name=None*, *continuation_ws=' '*)

Crée une instance `Header` à partir d'une séquence de paires renvoyées par `decode_header()`.

`decode_header()` prend une chaîne de valeur d'en-tête et renvoie une séquence de paires au format `(decoded_string, charset)` où *charset* est le nom du jeu de caractères.

Cette fonction prend l'une de ces séquences de paires et renvoie une instance `Header`. *maxlinelen*, *header_name* et *continuation_ws* (facultatifs) s'utilisent comme dans le constructeur `Header`.

20.1.12 email.charset : Representing character sets

Code source : Lib/email/charset.py

Ce module fait partie de l'ancienne API de messagerie (`Compat32`). Dans la nouvelle API, seule la table des alias est utilisée.

Le texte restant de cette section est la documentation originale de ce module.

Ce module fournit une classe `Charset` pour représenter les jeux de caractères et les conversions de jeux de caractères dans les messages électroniques, ainsi qu'un registre de jeux de caractères et plusieurs méthodes pratiques pour manipuler ce registre. Les instances de `Charset` sont utilisées dans plusieurs autres modules du paquet `email`.

Importez cette classe depuis le module `email.charset`.

class email.charset.**Charset**(*input_charset=DEFAULT_CHARSET*)

Associe les jeux de caractères à leurs propriétés d'e-mail.

Cette classe fournit des informations sur les exigences imposées aux e-mails pour un jeu de caractères spécifique. Elle fournit également des routines pratiques pour la conversion entre les jeux de caractères, compte tenu de la disponibilité des codecs applicables. Étant donné un jeu de caractères, elle fait de son mieux pour fournir des informations sur la façon d'utiliser ce jeu de caractères dans un message électronique d'une manière conforme à la RFC.

Certains jeux de caractères doivent être encodés avec *quoted-printable* ou en *base64* lorsqu'ils sont utilisés dans les en-têtes ou les corps des e-mails. Certains jeux de caractères doivent être convertis directement et ne sont pas autorisés dans les e-mails.

Le *input_charset* facultatif est tel que décrit ci-dessous ; il est toujours contraint en minuscules. Après avoir été normalisé par alias, il est également utilisé comme recherche dans le registre des jeux de caractères pour trouver le codage d'en-tête, le codage de corps et le codec de conversion de sortie à utiliser pour le jeu de caractères. Par exemple, si *input_charset* vaut `iso-8859-1`, alors les en-têtes et les corps seront encodés en utilisant *quoted-printable* et aucun codec de conversion de sortie n'est nécessaire. Si *input_charset* vaut `euc-jp`, alors les en-têtes seront encodés en *base64*, les corps ne seront pas encodés, mais le texte de sortie sera converti du jeu de caractères `euc-jp` vers le jeu de caractères `iso-2022-jp`.

Les instances de `Charset` ont les attributs de données suivants :

input_charset

Jeu de caractères initial spécifié. Les alias communs sont convertis en leurs noms de messagerie *officiels* (par exemple, `latin_1` est converti en `iso-8859-1`). Par défaut, `us-ascii` 7 bits.

header_encoding

If the character set must be encoded before it can be used in an email header, this attribute will be set to `charset.QP` (for quoted-printable), `charset.BASE64` (for base64 encoding), or `charset.SHORTEST` for the shortest of QP or BASE64 encoding. Otherwise, it will be `None`.

body_encoding

Same as *header_encoding*, but describes the encoding for the mail message's body, which indeed may be different than the header encoding. `charset.SHORTEST` is not allowed for *body_encoding*.

output_charset

Certains jeux de caractères doivent être convertis avant de pouvoir être utilisés dans les en-têtes ou le corps des e-mails. Si le *input_charset* est un jeu de ce type, alors cet attribut contient le nom du jeu de caractères vers lequel la sortie sera convertie. Sinon, il vaut `None`.

input_codec

Nom du codec Python utilisé pour convertir le *input_charset* en Unicode. Si aucun codec de conversion n'est nécessaire, cet attribut vaut `None`.

output_codec

Nom du codec Python utilisé pour convertir Unicode en *output_charset*. Si aucun codec de conversion n'est nécessaire, cet attribut a la même valeur que le *input_codec*.

Les instances de `Charset` ont également les méthodes suivantes :

get_body_encoding()

Renvoie l'encodage de transfert de contenu utilisé pour l'encodage du corps.

Il s'agit soit de la chaîne `quoted-printable` ou `base64` selon l'encodage utilisé, soit d'une fonction, vous devez alors appeler la fonction avec un seul argument, l'objet Message étant encodé. La fonction doit ensuite définir l'en-tête `Content-Transfer-Encoding` à la valeur appropriée.

Renvoie la chaîne `quoted-printable` si *body_encoding* est `QP`, renvoie la chaîne `base64` si *body_encoding* est `BASE64` et renvoie la chaîne `7bit` sinon.

get_output_charset()

Renvoie le jeu de caractères de sortie.

C'est l'attribut *output_charset* si ce n'est pas `None`, sinon c'est *input_charset*.

header_encode(*string*)

Encode la chaîne *string* pour un en-tête.

Le type d'encodage (*base64* ou *quoted-printable*) est basé sur l'attribut *header_encoding*.

header_encode_lines(*string*, *maxlengths*)

Encode *string* en la convertissant d'abord en octets, pour un en-tête.

C'est similaire à `header_encode()` sauf que la chaîne est adaptée aux longueurs de ligne maximales indiquées par l'argument *maxlengths*, qui doit être un itérateur : chaque élément renvoyé par cet itérateur fournit la prochaine longueur de ligne maximale.

body_encode(*string*)

Encode la chaîne *string* pour un usage en corps de message.

Le type d'encodage (*base64* ou *quoted-printable*) est basé sur l'attribut *body_encoding*.

La classe `Charset` fournit également un certain nombre de méthodes pour prendre en charge les opérations standard et les fonctions intégrées.

__str__()

Returns *input_charset* as a string coerced to lower case. `__repr__()` is an alias for `__str__()`.

__eq__(*other*)

Cette méthode vous permet de tester l'égalité de deux instances de `Charset`.

__ne__(*other*)

Cette méthode vous permet de tester l'inégalité de deux instances de `Charset`.

Le module `email.charset` fournit également les fonctions suivantes pour ajouter de nouvelles entrées à l'ensemble des jeux de caractères globaux, aux registres d'alias et de codec :

email.charset.**add_charset**(*charset*, *header_enc=None*, *body_enc=None*, *output_charset=None*)

Ajoute des propriétés relatives d'un jeu de caractères dans le registre global.

charset est le jeu de caractères d'entrée et doit être le nom canonique d'un jeu de caractères.

Optional *header_enc* and *body_enc* is either `charset.QP` for quoted-printable, `charset.BASE64` for base64 encoding, `charset.SHORTEST` for the shortest of quoted-printable or base64 encoding, or `None` for no encoding. `SHORTEST` is only valid for *header_enc*. The default is `None` for no encoding.

output_charset (facultatif) est le jeu de caractères dans lequel doit être la sortie. Les conversions se poursuivent du jeu de caractères d'entrée, vers Unicode, vers le jeu de caractères de sortie lorsque la méthode `Charset.convert()` est appelée. La valeur par défaut est de sortir dans le même jeu de caractères que l'entrée.

input_charset et *output_charset* doivent avoir des entrées de codec Unicode dans la table de correspondances du jeu de caractères au codec du module ; utilisez `add_codec()` pour ajouter des codecs que le module ne connaît pas. Voir la documentation du module `codecs` pour plus d'informations.

Le registre de jeux de caractères global est conservé dans le dictionnaire global du module `CHARSETS`.

email.charset.**add_alias**(*alias*, *canonical*)

Ajoute un alias de jeu de caractères. *alias* est le nom d'alias, par ex. `latin-1`. *canonical* est le nom canonique du jeu de caractères, par ex. `iso-8859-1`.

Le registre global des alias du jeu de caractères est conservé dans le dictionnaire global du module `ALIASES`.

email.charset.**add_codec**(*charset*, *codecname*)

Ajoute un codec qui fait correspondre les caractères du jeu de caractères donné vers et depuis Unicode.

charset est le nom canonique d'un jeu de caractères. *codecname* est le nom d'un codec Python (c.-à-d. une valeur valable comme second argument de la méthode `encode()` de `str`).

20.1.13 email.encoders : Encoders

Code source : Lib/email/encoders.py

Ce module fait partie du code patrimonial (`Compat32`) de l'API mail. Dans la nouvelle API la fonctionnalité est fournie par le paramètre *cte* de la méthode `set_content()`.

Ce module est obsolète en Python 3. Il n'est pas de besoin d'appeler explicitement les fonctions définies ici puisque la classe `MIMEText` ajuste le type de contenu et l'entête CTE à l'aide des paramètres *_subtype* et *_charset* de son constructeur.

Le texte restant de cette section est la documentation originale de ce module.

Au moment de la création d'objets `Message` à la main, il est souvent nécessaire d'encoder les charges utiles pour le transport à travers des serveurs mail conformes. C'est particulièrement vrai pour les messages de type `image/*` et `text/*` contenant des données binaires.

The `email` package provides some convenient encoders in its `encoders` module. These encoders are actually used by the `MIMEAudio` and `MIMEImage` class constructors to provide default encodings. All encoder functions take exactly one argument, the message object to encode. They usually extract the payload, encode it, and reset the payload to this newly encoded value. They should also set the `Content-Transfer-Encoding` header as appropriate.

À noter que ces fonctions n'ont pas de sens dans le cadre d'un message en plusieurs parties. Elles doivent à la place être appliquées aux sous-parties individuelles, et lèvent `TypeError` si on leur passe un message en plusieurs parties.

Voici les fonctions d'encodages fournies :

email.encoders.**encode_quopri**(*msg*)

Encode la charge utile au format Quoted-Printable, et assigne `quoted-printable`[1] à l'en-tête `Content-Transfer-Encoding`. C'est un bon encodage à utiliser quand la majorité de la charge utile contient essentiellement des données imprimables, à l'exceptions de quelques caractères.

1. À noter que l'encodage avec `encode_quopri()` encode également tous les caractères tabulation et espace.

email.encoders.**encode_base64**(*msg*)

Encode la charge utile au format *base64*, et assigne `base64` à l'en-tête `Content-Transfer-Encoding`. C'est un bon encodage à utiliser quand la majorité de la charge utile est non imprimable puisque c'est une forme plus compacte que *quoted-printable*.

email.encoders.**encode_7or8bit**(*msg*)

Ceci ne modifie pas effectivement la charge utile du message, mais va bien en revanche assigner la valeur `7bit` ou `8bit` à l'en-tête `Content-Transfer-Encoding` selon la nature de la charge utile.

email.encoders.**encode_noop**(*msg*)

Ceci ne fait rien ; et ne va même pas changer la valeur de l'en-tête `Content-Transfer-Encoding`.

Notes

20.1.14 email.utils : Miscellaneous utilities

Code source : Lib/email/utils.py

Il y a quelques utilitaires utiles fournis dans le module `email.utils` :

email.utils.**localtime**(*dt=None*)

Return local time as an aware datetime object. If called without arguments, return current time. Otherwise *dt* argument should be a `datetime` instance, and it is converted to the local time zone according to the system time zone database. If *dt* is naive (that is, `dt.tzinfo` is `None`), it is assumed to be in local time.

Ajouté dans la version 3.3.

Deprecated since version 3.12, removed in version 3.14 : The *isdst* parameter.

email.utils.**make_msgid**(*idstring=None*, *domain=None*)

Renvoie une chaîne adaptée à un en-tête `Message-ID` compatible **RFC 2822**. *idstring*, s'il est fourni, est une chaîne utilisée pour renforcer l'unicité de l'identifiant du message. Le *domain* facultatif, s'il est fourni, fournit la partie du *msgid* après le '@'. La valeur par défaut est le nom d'hôte local. Il n'est normalement pas nécessaire de remplacer cette valeur par défaut, mais cela peut être utile dans certains cas, comme la construction d'un système distribué qui utilise un nom de domaine cohérent sur plusieurs hôtes.

Modifié dans la version 3.2 : ajout du mot-clé *domaine*.

Les fonctions restantes font partie de l'ancienne API de messagerie (`Compat32`). Il n'est pas nécessaire de les utiliser directement avec la nouvelle API, car l'analyse et le formatage qu'elles fournissent sont effectués automatiquement par le mécanisme d'analyse d'en-tête de la nouvelle API.

email.utils.**quote**(*str*)

Renvoie une nouvelle chaîne avec les barres obliques inverses (*backslash*) dans *str* remplacées par deux barres obliques inverses et les guillemets doubles remplacés par des guillemets doubles échappés par une barre oblique inverse.

email.utils.**unquote**(*str*)

Renvoie une nouvelle chaîne qui est une version *sans guillemets* de *str*. Si *str* se termine et commence par des guillemets doubles, ils sont supprimés. De même, si *str* est encadrée par des chevrons mathématiques, ils sont supprimés.

email.utils.**parseaddr**(*address*, *, *strict=True*)

Analyse l'adresse – qui devrait être la valeur d'un champ contenant une adresse tel que `To` ou `Cc` – dans ses parties constituantes *realname* et *email address*. Renvoie un *n*-uplet de cette information, sauf si l'analyse échoue, alors un couple `('', '')` est renvoyé.

If *strict* is true, use a strict parser which rejects malformed inputs.

Modifié dans la version 3.13 : Add *strict* optional parameter and reject malformed inputs by default.

email.utils.**formataddr**(*pair*, *charset='utf-8'*)

L'inverse de `parseaddr()`, elle prend un couple de la forme `(realname, email_address)` et renvoie la valeur de chaîne appropriée pour un en-tête `To` ou `Cc`. Si le premier élément de *pair* s'évalue à faux, alors le deuxième élément est renvoyé sans modification.

Le *charset* optionnel est le jeu de caractères qui est utilisé dans l'encodage **RFC 2047** du `realname` si le `realname` contient des caractères non-ASCII. Cela peut être une instance de `str` ou un `Charset`. La valeur par défaut est `utf-8`.

Modifié dans la version 3.3 : ajout de l'option *charset*.

email.utils.**getaddresses**(*fieldvalues*, *, *strict=True*)

This method returns a list of 2-tuples of the form returned by `parseaddr()`. *fieldvalues* is a sequence of header field values as might be returned by `Message.get_all`.

If *strict* is true, use a strict parser which rejects malformed inputs.

Here's a simple example that gets all the recipients of a message :

```
from email.utils import getaddresses

tos = msg.get_all('to', [])
ccs = msg.get_all('cc', [])
resent_tos = msg.get_all('resent-to', [])
resent_ccs = msg.get_all('resent-cc', [])
all_recipients = getaddresses(tos + ccs + resent_tos + resent_ccs)
```

Modifié dans la version 3.13 : Add *strict* optional parameter and reject malformed inputs by default.

email.utils.**parsedate**(*date*)

Tente d'analyser une date selon les règles de la **RFC 2822**. Cependant, certains expéditeurs ne suivent pas ce format comme spécifié, donc `parsedate()` fait son possible dans de tels cas. *date* est une chaîne contenant une date **RFC 2822**, telle que `"Mon, 20 Nov 1995 19:12:08 -0500"`. Si elle réussit à analyser la date, `parsedate()` renvoie un 9-uplet qui peut être passé directement à `time.mktime()` ; sinon `None` est renvoyé. Notez que les indices 6, 7 et 8 du *n*-uplet résultat ne sont pas utilisables.

email.utils.**parsedate_tz**(*date*)

Effectue la même fonction que `parsedate()`, mais renvoie soit `None` ou un 10-uplet ; les 9 premiers éléments constituent un *n*-uplet qui peut être passé directement à `time.mktime()`, et le dixième est le décalage du fuseau horaire de la date par rapport à UTC (qui est le terme officiel pour *Greenwich Mean Time*)[1]. Si la chaîne d'entrée n'a pas de fuseau horaire, le dernier élément du *n*-uplet renvoyé est `0`, qui représente l'UTC. Notez que les indices 6, 7 et 8 du *n*-uplet résultat ne sont pas utilisables.

email.utils.**parsedate_to_datetime**(*date*)

L'inverse de `format_datetime()`. Effectue la même fonction que `parsedate()` mais, en cas de succès, renvoie un `datetime` ; sinon une `ValueError` est levée si *date* contient une valeur invalide telle qu'une heure supérieure à 23 ou un décalage de fuseau horaire non compris entre −24 et 24 heures. Si la date d'entrée a un fuseau horaire de `-0000`, le `datetime` sera un `datetime` naïf, et si la date est conforme aux RFC, elle représentera une heure en UTC mais sans indication du fuseau horaire source réel du message d'où provient la date. Si la date d'entrée a un autre décalage de fuseau horaire valide, le `datetime` sera un `datetime` avisé avec le `timezone tzinfo` adéquat.

Ajouté dans la version 3.3.

email.utils.**mktime_tz**(*tuple*)

Transforme un 10-uplet renvoyé par `parsedate_tz()` en un horodatage UTC (en secondes depuis *Epoch*). Si l'élément de fuseau horaire dans le *n*-uplet est `None`, elle considère que l'heure est locale.

email.utils.**formatdate**(*timeval=None*, *localtime=False*, *usegmt=False*)

Renvoie une chaîne de date conforme à la **RFC 2822**, par exemple :

1. Notez que le signe du décalage horaire est l'opposé du signe de la variable `time.timezone` pour le même fuseau horaire ; cette dernière variable suit la norme POSIX tandis que ce module suit la **RFC 2822**.

```
Fri, 09 Nov 2001 01:08:47 -0000
```

Optional *timeval* if given is a floating-point time value as accepted by `time.gmtime()` and `time.localtime()`, otherwise the current time is used.

L'option *localtime* est un indicateur qui, lorsqu'il est `True`, interprète *timeval* et renvoie une date relative au fuseau horaire local au lieu de UTC, en tenant correctement compte de l'heure d'été. La valeur par défaut est `False`, ce qui signifie que l'UTC est utilisée.

L'option *usegmt* est un indicateur qui, lorsqu'il est `True`, affiche une chaîne de date avec le fuseau horaire sous la forme de la chaîne ASCII `GMT`, plutôt qu'un `-0000` numérique. C'est nécessaire pour certains protocoles (tels que HTTP). Cela ne s'applique que lorsque *localtime* est `False`. La valeur par défaut est `False`.

`email.utils.`**`format_datetime`**(*dt*, *usegmt=False*)

Comme `formatdate`, mais l'entrée est une instance `datetime`. S'il s'agit d'un groupe date-heure naïf, il est considéré être « UTC sans information sur le fuseau horaire source », et le `-0000` conventionnel est utilisé pour le fuseau horaire. S'il s'agit d'un `datetime` avisé, alors le décalage numérique du fuseau horaire est utilisé. S'il s'agit d'un groupe date-heure avisé avec un décalage de fuseau horaire de zéro, alors *usegmt* peut être défini sur `True`, la chaîne `GMT` est alors utilisée à la place du décalage numérique du fuseau horaire. Cela fournit un moyen de générer des en-têtes de date HTTP conformes aux normes.

Ajouté dans la version 3.3.

`email.utils.`**`decode_rfc2231`**(*s*)

Décode la chaîne *s* selon la **RFC 2231**.

`email.utils.`**`encode_rfc2231`**(*s*, *charset=None*, *language=None*)

Encode la chaîne *s* selon la **RFC 2231**. *charset* et *language*, s'ils sont fournis, indiquent le jeu de caractères et le nom de la langue à utiliser. Si aucun n'est donné, *s* est renvoyé telle quelle. Si *charset* est donné mais que *language* ne l'est pas, la chaîne est encodée en utilisant la chaîne vide pour *language*.

`email.utils.`**`collapse_rfc2231_value`**(*value*, *errors='replace'*, *fallback_charset='us-ascii'*)

Lorsqu'un paramètre d'en-tête est encodé au format **RFC 2231**, `Message.get_param` peut renvoyer un triplet contenant le jeu de caractères, la langue et la valeur. `collapse_rfc2231_value()` transforme cela en une chaîne Unicode. *errors* (facultatif) est passé à l'argument *errors* de la méthode `encode()` de `str` ; sa valeur par défaut est `'replace'`. *fallback_charset* (facultatif) spécifie le jeu de caractères à utiliser si celui de l'en-tête **RFC 2231** n'est pas connu de Python ; sa valeur par défaut est `'us-ascii'`.

Pour plus de commodité, si la *valeur* transmise à `collapse_rfc2231_value()` n'est pas un *n*-uplet, ce doit être une chaîne et elle est renvoyée sans guillemets.

`email.utils.`**`decode_params`**(*params*)

Décode la liste des paramètres selon la **RFC 2231**. *params* est une séquence de couples contenant des éléments de la forme `(content-type, string-value)`.

Notes

20.1.15 `email.iterators` : Iterators

Code source : Lib/email/iterators.py

Itérer sur l'arborescence d'un objet message est plutôt simple avec la méthode `Message.walk`. Le module `email.iterators` fournit des fonctionnalités d'itérations de plus haut niveau sur les arbres d'objets messages.

`email.iterators.`**`body_line_iterator`**(*msg*, *decode=False*)

Cette fonction permet d'itérer sur tous les contenus de tous les éléments de *msg*, en retournant les contenus sous forme de chaînes de caractères ligne par ligne. Il saute les entêtes des sous éléments, et tous les sous éléments dont le contenu n'est pas une chaîne de caractères Python. C'est en quelque sorte équivalent à une lecture plate d'une représentation textuelle du message à partir d'un fichier en utilisant `readline()`, et en sautant toutes les entêtes intermédiaires.

Le paramètre optionnel *decode* est transmis à la méthode `Message.get_payload`.

email.iterators.**typed_subpart_iterator**(*msg*, *maintype='text'*, *subtype=None*)

Cette fonction permet d'itérer sur tous les sous éléments de *msg*, en retournant seulement les sous éléments qui correspondent au type MIME spécifié par *maintype* et *subtype*.

Notez que le paramètre *subtype* est optionnel ; s'il n'est pas présent, alors le type MIME du sous élément est seulement composé du type principal. *maintype* est également optionnel ; sa valeur par défaut est `text`.

En conséquence, par défaut, `typed_subpart_iterator()` retourne chaque sous élément qui a un type MIME de type `text/*`.

La fonction suivante a été ajouté en tant qu'un outil de débogage. Elle *ne devrait pas* être considérée comme une interface publique supportée pour ce paquet.

email.iterators.**_structure**(*msg*, *fp=None*, *level=0*, *include_default=False*)

Affiche une représentation indentée des types de contenu de la structure de l'objet message. Par exemple :

```
>>> msg = email.message_from_file(somefile)
>>> _structure(msg)
multipart/mixed
    text/plain
    text/plain
    multipart/digest
        message/rfc822
            text/plain
        message/rfc822
            text/plain
        message/rfc822
            text/plain
        message/rfc822
            text/plain
        message/rfc822
            text/plain
    text/plain
```

Le paramètre optionnel *fp* est un objet fichier-compatible dans lequel on peut écrire le flux de sortie. Il doit être approprié pour la fonction de Python `print()`. *level* est utilisé en interne. *include_default*, si vrai, affiche aussi le type par défaut.

Voir aussi

Module `smtplib`
Client SMTP (*Simple Mail Transport Protocol*)

Module `poplib`
Client POP (*Post Office Protocol*)

Module `imaplib`
Client IMAP (*Internet Message Access Protocol*)

Module `mailbox`
Outils pour créer, lire et gérer des messages regroupés sur disque en utilisant des formats standards variés.

20.2 `json` --- JSON encoder and decoder

Code source : Lib/json/__init__.py

JSON (JavaScript Object Notation), specified by **RFC 7159** (which obsoletes **RFC 4627**) and by ECMA-404, is a lightweight data interchange format inspired by JavaScript object literal syntax (although it is not a strict subset of

JavaScript [1]).

Avertissement

Be cautious when parsing JSON data from untrusted sources. A malicious JSON string may cause the decoder to consume considerable CPU and memory resources. Limiting the size of data to be parsed is recommended.

json fournit une API familière aux utilisateurs des modules *marshal* et *pickle* de la bibliothèque standard.

Encodage de quelques objets de base Python :

```
>>> import json
>>> json.dumps(['foo', {'bar': ('baz', None, 1.0, 2)}])
'["foo", {"bar": ["baz", null, 1.0, 2]}]'
>>> print(json.dumps("\"foo\bar"))
"\"foo\bar"
>>> print(json.dumps('\u1234'))
"\u1234"
>>> print(json.dumps('\\'))
"\\"
>>> print(json.dumps({"c": 0, "b": 0, "a": 0}, sort_keys=True))
{"a": 0, "b": 0, "c": 0}
>>> from io import StringIO
>>> io = StringIO()
>>> json.dump(['streaming API'], io)
>>> io.getvalue()
'["streaming API"]'
```

Encodage compact :

```
>>> import json
>>> json.dumps([1, 2, 3, {'4': 5, '6': 7}], separators=(',', ':'))
'[1,2,3,{"4":5,"6":7}]'
```

Affichage plus lisible :

```
>>> import json
>>> print(json.dumps({'6': 7, '4': 5}, sort_keys=True, indent=4))
{
    "4": 5,
    "6": 7
}
```

Specializing JSON object encoding :

```
>>> import json
>>> def custom_json(obj):
...     if isinstance(obj, complex):
...         return {'__complex__': True, 'real': obj.real, 'imag': obj.imag}
...     raise TypeError(f'Cannot serialize object of {type(obj)}')
...
>>> json.dumps(1 + 2j, default=custom_json)
'{"__complex__": true, "real": 1.0, "imag": 2.0}'
```

Décodage JSON :

1. Comme noté dans l'errata de la RFC 7159, JSON autorise les caractères littéraux U+2028 (*LINE SEPARATOR*) et U+2029 (*PARAGRAPH SEPARATOR*) dans les chaînes de caractères, alors que Javascript (selon le standard ECMAScript édition 5.1) ne le permet pas.

```
>>> import json
>>> json.loads('["foo", {"bar":["baz", null, 1.0, 2]}]')
['foo', {'bar': ['baz', None, 1.0, 2]}]
>>> json.loads('"\\"foo\\bar"')
'"foo\x08ar'
>>> from io import StringIO
>>> io = StringIO('["streaming API"]')
>>> json.load(io)
['streaming API']
```

Spécialisation du décodage JSON pour un objet :

```
>>> import json
>>> def as_complex(dct):
...     if '__complex__' in dct:
...         return complex(dct['real'], dct['imag'])
...     return dct
...
>>> json.loads('{"__complex__": true, "real": 1, "imag": 2}',
...     object_hook=as_complex)
(1+2j)
>>> import decimal
>>> json.loads('1.1', parse_float=decimal.Decimal)
Decimal('1.1')
```

Extension de la classe `JSONEncoder` :

```
>>> import json
>>> class ComplexEncoder(json.JSONEncoder):
...     def default(self, obj):
...         if isinstance(obj, complex):
...             return [obj.real, obj.imag]
...         # Let the base class default method raise the TypeError
...         return super().default(obj)
...
>>> json.dumps(2 + 1j, cls=ComplexEncoder)
'[2.0, 1.0]'
>>> ComplexEncoder().encode(2 + 1j)
'[2.0, 1.0]'
>>> list(ComplexEncoder().iterencode(2 + 1j))
['[2.0', ', 1.0', ']']
```

Using `json` from the shell to validate and pretty-print :

```
$ echo '{"json":"obj"}' | python -m json
{
    "json": "obj"
}
$ echo '{1.2:3.4}' | python -m json
Expecting property name enclosed in double quotes: line 1 column 2 (char 1)
```

Voir *Command-line interface* pour une documentation détaillée.

Note

JSON is a subset of YAML 1.2. The JSON produced by this module's default settings (in particular, the default *separators* value) is also a subset of YAML 1.0 and 1.1. This module can thus also be used as a YAML serializer.

> **Note**
>
> Les encodeurs et décodeurs de ce module conservent l'ordre d'entrée et de sortie par défaut. L'ordre n'est perdu que si les conteneurs sous-jacents ne sont pas ordonnés.

20.2.1 Utilisation de base

`json.dump`(*obj*, *fp*, ***, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *cls=None*, *indent=None*, *separators=None*, *default=None*, *sort_keys=False*, ***kw*)

Serialize *obj* as a JSON formatted stream to *fp* (a `.write()`-supporting *file-like object*) using this *Python-to-JSON conversion table*.

> **Note**
>
> À l'inverse de `pickle` et `marshal`, JSON n'est pas un protocole par trames, donc essayer de sérialiser plusieurs objets par des appels répétés à `dump()` en utilisant le même *fp* résulte en un fichier JSON invalide.

Paramètres

- **obj** (`object`) -- The Python object to be serialized.
- **fp** (*file-like object*) -- The file-like object *obj* will be serialized to. The `json` module always produces `str` objects, not `bytes` objects, therefore `fp.write()` must support `str` input.
- **skipkeys** (`bool`) -- If `True`, keys that are not of a basic type (`str`, `int`, `float`, `bool`, `None`) will be skipped instead of raising a `TypeError`. Default `False`.
- **ensure_ascii** (`bool`) -- If `True` (the default), the output is guaranteed to have all incoming non-ASCII characters escaped. If `False`, these characters will be outputted as-is.
- **check_circular** (`bool`) -- If `False`, the circular reference check for container types is skipped and a circular reference will result in a `RecursionError` (or worse). Default `True`.
- **allow_nan** (`bool`) -- If `False`, serialization of out-of-range `float` values (`nan`, `inf`, `-inf`) will result in a `ValueError`, in strict compliance with the JSON specification. If `True` (the default), their JavaScript equivalents (`NaN`, `Infinity`, `-Infinity`) are used.
- **cls** (a `JSONEncoder` subclass) -- If set, a custom JSON encoder with the `default()` method overridden, for serializing into custom datatypes. If `None` (the default), `JSONEncoder` is used.
- **indent** (`int` | `str` | *None*) -- If a positive integer or string, JSON array elements and object members will be pretty-printed with that indent level. A positive integer indents that many spaces per level ; a string (such as `"\t"`) is used to indent each level. If zero, negative, or `""` (the empty string), only newlines are inserted. If `None` (the default), the most compact representation is used.
- **separators** (`tuple` | *None*) -- A two-tuple : `(item_separator, key_separator)`. If `None` (the default), *separators* defaults to `(', ', ': ')` if *indent* is `None`, and `(',', ': ')` otherwise. For the most compact JSON, specify `(',', ':')` to eliminate whitespace.
- **default** (*callable* | None) -- A function that is called for objects that can't otherwise be serialized. It should return a JSON encodable version of the object or raise a `TypeError`. If `None` (the default), `TypeError` is raised.
- **sort_keys** (`bool`) -- If `True`, dictionaries will be outputted sorted by key. Default `False`.

Modifié dans la version 3.2 : Autorise les chaînes en plus des nombres entiers pour *indent*.

Modifié dans la version 3.4 : Utilise `(',', ': ')` par défaut si *indent* n'est pas `None`.

Modifié dans la version 3.6 : tous les paramètres optionnels sont maintenant des *keyword-only*.

`json.dumps`(*obj*, ***, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *cls=None*, *indent=None*, *separators=None*, *default=None*, *sort_keys=False*, ***kw*)

Sérialise *obj* en une `str` formatée en JSON, en utilisant cette *table de conversion*. Les arguments ont la même

signification que ceux de `dump()`.

> **Note**
>
> Les clés dans les couples JSON clé-valeur sont toujours de type `str`. Quand un dictionnaire est converti en JSON, toutes les clés du dictionnaire sont transformées en chaînes de caractères. Ce qui fait que si un dictionnaire est converti en JSON et reconverti en dictionnaire, le résultat peut ne pas être égal à l'original. Ainsi, `loads(dumps(x)) != x` si x contient des clés qui ne sont pas des chaînes.

`json.load`(*fp*, *, *cls=None*, *object_hook=None*, *parse_float=None*, *parse_int=None*, *parse_constant=None*, *object_pairs_hook=None*, ***kw*)

Deserialize *fp* to a Python object using the *JSON-to-Python conversion table*.

Paramètres

- **fp** (*file-like object*) -- A `.read()`-supporting *text file* or *binary file* containing the JSON document to be deserialized.
- **cls** (a `JSONDecoder` subclass) -- If set, a custom JSON decoder. Additional keyword arguments to `load()` will be passed to the constructor of *cls*. If `None` (the default), `JSONDecoder` is used.
- **object_hook** (*callable* | None) -- If set, a function that is called with the result of any object literal decoded (a `dict`). The return value of this function will be used instead of the `dict`. This feature can be used to implement custom decoders, for example JSON-RPC class hinting. Default `None`.
- **object_pairs_hook** (*callable* | None) -- If set, a function that is called with the result of any object literal decoded with an ordered list of pairs. The return value of this function will be used instead of the `dict`. This feature can be used to implement custom decoders. If *object_hook* is also set, *object_pairs_hook* takes priority. Default `None`.
- **parse_float** (*callable* | None) -- If set, a function that is called with the string of every JSON float to be decoded. If `None` (the default), it is equivalent to `float(num_str)`. This can be used to parse JSON floats into custom datatypes, for example `decimal.Decimal`.
- **parse_int** (*callable* | None) -- If set, a function that is called with the string of every JSON int to be decoded. If `None` (the default), it is equivalent to `int(num_str)`. This can be used to parse JSON integers into custom datatypes, for example `float`.
- **parse_constant** (*callable* | None) -- If set, a function that is called with one of the following strings : `'-Infinity'`, `'Infinity'`, or `'NaN'`. This can be used to raise an exception if invalid JSON numbers are encountered. Default `None`.

Lève

- ***JSONDecodeError*** -- When the data being deserialized is not a valid JSON document.
- ***UnicodeDecodeError*** -- When the data being deserialized does not contain UTF-8, UTF-16 or UTF-32 encoded data.

Modifié dans la version 3.1 :
- Added the optional *object_pairs_hook* parameter.
- *parse_constant* n'est plus appelée pour *null*, *true* ou *false*.

Modifié dans la version 3.6 :
- tous les paramètres optionnels sont maintenant des *keyword-only*.
- *fp* peut maintenant être un `binary file`. Son encodage doit être UTF-8, UTF-16 ou UTF-32.

Modifié dans la version 3.11 : The default *parse_int* of `int()` now limits the maximum length of the integer string via the interpreter's *integer string conversion length limitation* to help avoid denial of service attacks.

`json.loads`(*s*, *, *cls=None*, *object_hook=None*, *parse_float=None*, *parse_int=None*, *parse_constant=None*, *object_pairs_hook=None*, ***kw*)

Identical to `load()`, but instead of a file-like object, deserialize *s* (a `str`, `bytes` or `bytearray` instance containing a JSON document) to a Python object using this *conversion table*.

Modifié dans la version 3.6 : *s* peut maintenant être de type `bytes` ou `bytearray`. L'encodage d'entrée doit être UTF-8, UTF-16 ou UTF-32.

Modifié dans la version 3.9 : suppression de l'argument nommé *encoding*.

20.2.2 Encodeurs et décodeurs

class `json.JSONDecoder`(*, *object_hook=None*, *parse_float=None*, *parse_int=None*, *parse_constant=None*, *strict=True*, *object_pairs_hook=None*)

Décodeur simple JSON.

Applique par défaut les conversions suivantes en décodant :

JSON	Python
objet	*dict*
array	*list*
string	*str*
number (nombre entier)	*int*
number (nombre réel)	*float*
true	*True*
false	*False*
null	`None`

Les valeurs `NaN`, `Infinity` et `-Infinity` sont aussi comprises comme leurs valeurs `float` correspondantes, bien que ne faisant pas partie de la spécification JSON.

object_hook is an optional function that will be called with the result of every JSON object decoded and its return value will be used in place of the given `dict`. This can be used to provide custom deserializations (e.g. to support JSON-RPC class hinting).

object_pairs_hook is an optional function that will be called with the result of every JSON object decoded with an ordered list of pairs. The return value of *object_pairs_hook* will be used instead of the `dict`. This feature can be used to implement custom decoders. If *object_hook* is also defined, the *object_pairs_hook* takes priority.

Modifié dans la version 3.1 : ajout du support de *object_pairs_hook*.

parse_float is an optional function that will be called with the string of every JSON float to be decoded. By default, this is equivalent to `float(num_str)`. This can be used to use another datatype or parser for JSON floats (e.g. `decimal.Decimal`).

parse_int is an optional function that will be called with the string of every JSON int to be decoded. By default, this is equivalent to `int(num_str)`. This can be used to use another datatype or parser for JSON integers (e.g. `float`).

parse_constant is an optional function that will be called with one of the following strings : `'-Infinity'`, `'Infinity'`, `'NaN'`. This can be used to raise an exception if invalid JSON numbers are encountered.

Si *strict* vaut `False` (`True` par défaut), alors les caractères de contrôle sont autorisés à l'intérieur des chaînes. Les caractères de contrôle dans ce contexte sont ceux dont les codes sont dans l'intervalle 0--31, incluant `'\t'` (tabulation), `'\n'`, `'\r'` et `'\0'`.

Si les données à désérialiser ne sont pas un document JSON valide, une `JSONDecodeError` est levée.

Modifié dans la version 3.6 : Tous les paramètres sont maintenant des *keyword-only*.

decode(*s*)

Renvoie la représentation Python de *s* (une instance `str` contenant un document JSON).

Une `JSONDecodeError` est levée si le document JSON donné n'est pas valide.

raw_decode(*s*)

Décode en document JSON depuis *s* (une instance `str` débutant par un document JSON) et renvoie un *n*-uplet de 2 éléments contenant la représentation Python de l'objet et l'index dans *s* où le document se terminait.

Elle peut être utilisée pour décoder un document JSON depuis une chaîne qui peut contenir des données supplémentaires à la fin.

class `json.JSONEncoder`(*, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *default=None*)

Encodeur JSON extensible pour les structures de données Python.

Prend en charge par défaut les objets et types suivants :

Python	JSON
dict	objet
list, *tuple*	*array*
str	*string*
int, *float*, et *Enums* dérivées d'*int* ou de *float*	*number*
True	*true*
False	*false*
`None`	*null*

Modifié dans la version 3.4 : ajout de la prise en charge des classes *Enum* dérivées d'*int* ou de *float*.

To extend this to recognize other objects, subclass and implement a `default()` method with another method that returns a serializable object for `o` if possible, otherwise it should call the superclass implementation (to raise `TypeError`).

Si *skipkeys* vaut `False` (valeur par défaut), une `TypeError` est levée si la clé encodée n'est pas de type `str`, `int`, `float` ou `None`. Si *skipkeys* vaut `True`, ces éléments sont simplement ignorés.

Si *ensure_ascii* vaut `True` (valeur par défaut), les caractères non ASCII sont tous échappés à la sortie. Si *ensure_ascii* vaut `False`, ils sont écrits tels quels.

Si *check_circular* vaut `True` (valeur par défaut), une vérification a lieu sur les listes, dictionnaires et objets personnalisés, afin de détecter les références circulaires et éviter les récursions infinies (qui causeraient une `RecursionError`). Autrement, la vérification n'a pas lieu.

Si *allow_nan* vaut `True` (valeur par défaut), alors `NaN`, `Infinity` et `-Infinity` sont encodés comme tels. Ce comportement ne respecte pas la spécification JSON, mais est cohérent avec le majorité des encodeurs-décodeurs JavaScript. Autrement, une `ValueError` est levée pour de telles valeurs.

Si *sort_keys* vaut `True` (`False` par défaut), alors les dictionnaires sont triés par clés en sortie ; cela est utile lors de tests de régression pour pouvoir comparer les sérialisations JSON au jour le jour.

Si *indent* est un nombre entier positif ou une chaîne de caractères, les éléments de tableaux et les membres d'objets JSON sont affichés élégamment avec ce niveau d'indentation. Un niveau d'indentation de 0, négatif, ou `""` n'insère que des retours à la ligne. `None` (la valeur par défaut) choisit la représentation la plus compacte. Utiliser un entier positif pour *indent* indente d'autant d'espaces par niveau. Si *indent* est une chaîne (telle que `"\t"`), cette chaîne est utilisée pour indenter à chaque niveau.

Modifié dans la version 3.2 : Autorise les chaînes en plus des nombres entiers pour *indent*.

Si spécifié, *separators* doit être un *n*-uplet `(item_separator, key_separator)`. Sa valeur par défaut est `(', ', ': ')` si *indent* est `None`, et `(',', ': ')` autrement. Pour obtenir la représentation JSON la plus compacte possible, passez `(',', ':')` pour éliminer les espaces.

Modifié dans la version 3.4 : Utilise `(',', ': ')` par défaut si *indent* n'est pas `None`.

Si spécifié, *default* doit être une fonction qui sera appelée pour les objets qui ne peuvent être sérialisés autrement. Elle doit renvoyer une représentation composée d'objets Python sérialisable en JSON ou lever une `TypeError`. Si non spécifié, une `TypeError` est levée.

Modifié dans la version 3.6 : Tous les paramètres sont maintenant des *keyword-only*.

default(*o*)

Implémentez cette méthode dans une sous-classe afin qu'elle renvoie un objet sérialisable pour *o*, ou appelle l'implémentation de base (qui lèvera une `TypeError`).

For example, to support arbitrary iterators, you could implement `default()` like this :

```
def default(self, o):
    try:
        iterable = iter(o)
    except TypeError:
        pass
    else:
        return list(iterable)
    # Let the base class default method raise the TypeError
    return super().default(o)
```

encode(*o*)

Renvoie une chaîne JSON représentant la structure de données Python *o*. Par exemple :

```
>>> json.JSONEncoder().encode({"foo": ["bar", "baz"]})
'{"foo": ["bar", "baz"]}'
```

iterencode(*o*)

Encode l'objet *o* donné, et produit chaque chaîne représentant l'objet selon disponibilité. Par exemple :

```
for chunk in json.JSONEncoder().iterencode(bigobject):
    mysocket.write(chunk)
```

20.2.3 Exceptions

exception `json.`**JSONDecodeError**(*msg*, *doc*, *pos*)

Sous-classe de `ValueError` avec les attributs additionnels suivants :

msg

Le message d'erreur non formaté.

doc

Le document JSON en cours de traitement.

pos

L'index de *doc* à partir duquel l'analyse a échoué.

lineno

La ligne correspondant à *pos*.

colno

La colonne correspondant à *pos*.

Ajouté dans la version 3.5.

20.2.4 Conformité au standard et Interopérabilité

The JSON format is specified by **RFC 7159** and by ECMA-404. This section details this module's level of compliance with the RFC. For simplicity, `JSONEncoder` and `JSONDecoder` subclasses, and parameters other than those explicitly mentioned, are not considered.

Ce module ne se conforme pas strictement à la RFC, implémentant quelques extensions qui sont valides en JavaScript mais pas en JSON. En particulier :

— Les nombres infinis et *NaN* sont acceptés et retranscrits ;

— Les noms répétés au sein d'un objet sont acceptés, seule la valeur du dernier couple nom-valeur est utilisée.

Comme la RFC permet aux analyseurs conformes d'accepter en entrée des textes non conformes, le désérialiseur de ce module avec ses paramètres par défaut est techniquement conforme à la RFC.

Encodage des caractères

La RFC requiert que le JSON soit représenté en utilisant l'encodage UTF-8, UTF-16 ou UTF-32, avec UTF-8 recommandé par défaut pour une interopérabilité maximale.

Comme cela est permis par la RFC, bien que non requis, le sérialiseur du module active *ensure_ascii=True* par défaut, échappant ainsi la sortie de façon à ce que les chaînes résultants ne contiennent que des caractères ASCII.

Outre le paramètre *ensure_ascii*, les conversions entre objets Python et `chaînes Unicode` de ce module sont strictement définies, et ne résolvent donc pas directement le problème de l'encodage des caractères.

La RFC interdit d'ajouter un octet marqueur d'ordre (*byte mark order* ou BOM) au début du texte JSON, et le sérialiseur de ce module n'ajoute pas de tel BOM. La RFC permet, mais ne requiert pas, que les désérialiseurs JSON ignorent ces BOM. Le désérialiseur de ce module lève une `ValueError` quand un BOM est présent au début du fichier.

La RFC n'interdit pas explicitement les chaînes JSON contenant des séquences d'octets ne correspondant à aucun caractère Unicode valide (p. ex. les *surrogates* UTF-16 sans correspondance), mais précise que cela peut causer des problèmes d'interopérabilité. Par défaut, ce module accepte et retranscrit (quand présents dans la `str` originale) les *code points* de telles séquences.

Valeurs numériques infinies et NaN

La RFC ne permet pas la représentation des nombres infinis ou des *NaN*. Néanmoins, par défaut, ce module accepte et retranscrit `Infinity`, `-Infinity` et `NaN` comme s'ils étaient des valeurs numériques littérales JSON valides :

```
>>> # Neither of these calls raises an exception, but the results are not valid␣
↪JSON
>>> json.dumps(float('-inf'))
'-Infinity'
>>> json.dumps(float('nan'))
'NaN'
>>> # Same when deserializing
>>> json.loads('-Infinity')
-inf
>>> json.loads('NaN')
nan
```

Dans le sérialiseur, le paramètre *allow_nan* peut être utilisé pour altérer ce comportement. Dans le désérialiseur, le paramètre *parse_constant* peut être utilisé pour changer ce comportement.

Noms répétés au sein d'un objet

La RFC précise que les noms au sein d'un objet JSON doivent être uniques, mais ne décrit pas comment les noms répétés doivent être gérés. Par défaut, ce module ne lève pas d'exception ; à la place, il ignore tous les couples nom-valeur sauf le dernier pour un nom donné :

```
>>> weird_json = '{"x": 1, "x": 2, "x": 3}'
>>> json.loads(weird_json)
{'x': 3}
```

Le paramètre *object_pairs_hook* peut être utilisé pour modifier ce comportement.

Valeurs de plus haut niveau (hors objets ou tableaux)

L'ancienne version de JSON définie par l'obsolète **RFC 4627** demandait à ce que la valeur de plus haut niveau du texte JSON soit un objet ou un tableau JSON (`dict` ou `list` Python), et ne soit pas *null*, un nombre, ou une chaîne de caractères. La **RFC 7159** a supprimé cette restriction, jamais implémentée par ce module, que ce soit dans le sérialiseur ou le désérialiseur.

Cependant, pour une interopérabilité maximale, vous pourriez volontairement souhaiter adhérer à cette restriction.

Limitations de l'implémentation

Certaines implémentations de désérialiseurs JSON peuvent avoir des limites sur :

- la taille des textes JSON acceptés ;
- le niveau maximum d'objets et tableaux JSON imbriqués ;
- l'intervalle et la précision des nombres JSON ;
- le contenu et la longueur maximale des chaînes JSON.

Ce module n'impose pas de telles limites si ce n'est celles inhérentes aux types de données Python ou à l'interpréteur.

Lors d'une sérialisation JSON, faites attention à ces limitations dans les applications qui utilisent votre JSON. En particulier, il est courant pour les nombres JSON d'être désérialisés vers des nombres IEEE 754 à précision double, et donc sujets à l'intervalle et aux limitations sur la précision de cette représentation. Cela est d'autant plus important lors de la sérialisation de grands `int` Python, ou d'instances de types numériques « exotiques » comme `decimal.Decimal`.

20.2.5 Command-line interface

Code source : Lib/json/tool.py

The `json` module can be invoked as a script via `python -m json` to validate and pretty-print JSON objects. The `json.tool` submodule implements this interface.

If the optional `infile` and `outfile` arguments are not specified, `sys.stdin` and `sys.stdout` will be used respectively :

```
$ echo '{"json": "obj"}' | python -m json
{
    "json": "obj"
}
$ echo '{1.2:3.4}' | python -m json
Expecting property name enclosed in double quotes: line 1 column 2 (char 1)
```

Modifié dans la version 3.5 : La sortie conserve maintenant l'ordre des données de l'entrée. Utilisez l'option `--sort-keys` pour sortir des dictionnaires triés alphabétiquement par clés.

Modifié dans la version 3.14 : The `json` module may now be directly executed as `python -m json`. For backwards compatibility, invoking the CLI as `python -m json.tool` remains supported.

Command-line options

`infile`

Le fichier JSON à valider ou réécrire élégamment :

```
$ python -m json mp_films.json
[
    {
        "title": "And Now for Something Completely Different",
        "year": 1971
    },
    {
        "title": "Monty Python and the Holy Grail",
        "year": 1975
    }
]
```

If *infile* is not specified, read from `sys.stdin`.

`outfile`

Write the output of the *infile* to the given *outfile*. Otherwise, write it to `sys.stdout`.

--sort-keys

Trie alphabétiquement les dictionnaires par clés.

Ajouté dans la version 3.5.

--no-ensure-ascii

Désactive l'échappement des caractères non ASCII, voir `json.dumps()` pour plus d'informations.

Ajouté dans la version 3.9.

--json-lines

Transforme chaque ligne d'entrée en un objet JSON individuel.

Ajouté dans la version 3.8.

--indent, --tab, --no-indent, --compact

Options mutuellement exclusives de contrôle des espaces.

Ajouté dans la version 3.9.

-h, --help

Affiche le message d'aide.

Notes

20.3 `mailbox` --- Manipulate mailboxes in various formats

Code source : Lib/mailbox.py

This module defines two classes, `Mailbox` and `Message`, for accessing and manipulating on-disk mailboxes and the messages they contain. `Mailbox` offers a dictionary-like mapping from keys to messages. `Message` extends the `email.message` module's `Message` class with format-specific state and behavior. Supported mailbox formats are Maildir, mbox, MH, Babyl, and MMDF.

Voir aussi

Module `email`
Représente et manipule des messages.

20.3.1 `Mailbox` objects

class mailbox.**Mailbox**

Une boîte mail, qui peut être inspectée et modifiée.

The `Mailbox` class defines an interface and is not intended to be instantiated. Instead, format-specific subclasses should inherit from `Mailbox` and your code should instantiate a particular subclass.

The `Mailbox` interface is dictionary-like, with small keys corresponding to messages. Keys are issued by the `Mailbox` instance with which they will be used and are only meaningful to that `Mailbox` instance. A key continues to identify a message even if the corresponding message is modified, such as by replacing it with another message.

Messages may be added to a `Mailbox` instance using the set-like method `add()` and removed using a `del` statement or the set-like methods `remove()` and `discard()`.

`Mailbox` interface semantics differ from dictionary semantics in some noteworthy ways. Each time a message is requested, a new representation (typically a `Message` instance) is generated based upon the current state of the mailbox. Similarly, when a message is added to a `Mailbox` instance, the provided message representation's contents are copied. In neither case is a reference to the message representation kept by the `Mailbox` instance.

The default `Mailbox` *iterator* iterates over message representations, not keys as the default `dictionary` iterator does. Moreover, modification of a mailbox during iteration is safe and well-defined. Messages added to the mailbox after an iterator is created will not be seen by the iterator. Messages removed from the mailbox before the iterator yields them will be silently skipped, though using a key from an iterator may result in a `KeyError` exception if the corresponding message is subsequently removed.

> **Avertissement**
>
> Be very cautious when modifying mailboxes that might be simultaneously changed by some other process. The safest mailbox format to use for such tasks is `Maildir`; try to avoid using single-file formats such as `mbox` for concurrent writing. If you're modifying a mailbox, you *must* lock it by calling the `lock()` and `unlock()` methods *before* reading any messages in the file or making any changes by adding or deleting a message. Failing to lock the mailbox runs the risk of losing messages or corrupting the entire mailbox.

`Mailbox` instances have the following methods :

add(*message*)

Ajoute *message* à la boîte mail et renvoie la clé qui lui a été assigné.

Le paramètre *message* peut être une instance `Message`, une instance `email.message.Message`, une chaîne de caractères, une séquence d'octets ou un objet fichier-compatible (qui doit être ouvert en mode binaire). Si *message* est une instance de la sous-classe `Message` au format correspondant (par exemple s'il s'agit d'une instance `mboxMessage` et d'une instance `mbox`), les informations spécifiques à son format sont utilisées. Sinon, des valeurs par défaut raisonnables pour son format sont utilisées.

Modifié dans la version 3.2 : Ajout de la gestion des messages binaires.

remove(*key*)

__delitem__(*key*)

discard(*key*)

Supprime le message correspondant à *key* dans la boîte mail.

Si ce message n'existe pas, une exception `KeyError` est levée si la méthode a été appelée en tant que `remove()` ou `__delitem__()` ; mais aucune exception n'est levée si la méthode a été appelée en tant que `discard()`. Vous préférerez sûrement le comportement de `discard()` si le format de boîte mail sous-jacent accepte la modification concurrente par les autres processus.

__setitem__(*key*, *message*)

Remplace le message correspondant à *key* par *message*. Lève une exception `KeyError` s'il n'y a pas déjà de message correspondant à *key*.

Comme pour `add()`, le paramètre *message* peut être une instance `Message`, une instance `email.message.Message`, une chaîne de caractères, une chaîne d'octets ou un objet fichier-compatible (qui doit être ouvert en mode binaire). Si *message* est une instance de la sous-classe `Message` au format correspondant (par exemple s'il s'agit d'une instance `mboxMessage` et d'une instance `mbox`), les informations spécifiques à son format sont utilisées. Sinon, les informations spécifiques au format du message qui correspond à *key* ne sont modifiées.

iterkeys()

Return an *iterator* over all keys

keys()

The same as `iterkeys()`, except that a `list` is returned rather than an *iterator*

itervalues()

__iter__()

Return an *iterator* over representations of all messages. The messages are represented as instances of the appropriate format-specific `Message` subclass unless a custom message factory was specified when the `Mailbox` instance was initialized.

> **Note**
>
> Le comportement de `__iter__()` diffère de celui d'un dictionnaire, pour lequel l'itération se fait sur ses clés.

values()

The same as `itervalues()`, except that a `list` is returned rather than an *iterator*

iteritems()

Return an *iterator* over (*key*, *message*) pairs, where *key* is a key and *message* is a message representation. The messages are represented as instances of the appropriate format-specific `Message` subclass unless a custom message factory was specified when the `Mailbox` instance was initialized.

items()

The same as `iteritems()`, except that a `list` of pairs is returned rather than an *iterator* of pairs.

get(*key*, *default=None*)

__getitem__(*key*)

Return a representation of the message corresponding to *key*. If no such message exists, *default* is returned if the method was called as `get()` and a `KeyError` exception is raised if the method was called as `__getitem__()`. The message is represented as an instance of the appropriate format-specific `Message` subclass unless a custom message factory was specified when the `Mailbox` instance was initialized.

get_message(*key*)

Return a representation of the message corresponding to *key* as an instance of the appropriate format-specific `Message` subclass, or raise a `KeyError` exception if no such message exists.

get_bytes(*key*)

Return a byte representation of the message corresponding to *key*, or raise a `KeyError` exception if no such message exists.

Ajouté dans la version 3.2.

get_string(*key*)

Return a string representation of the message corresponding to *key*, or raise a `KeyError` exception if no such message exists. The message is processed through `email.message.Message` to convert it to a 7bit clean representation.

get_file(*key*)

Return a *file-like* representation of the message corresponding to *key*, or raise a `KeyError` exception if no such message exists. The file-like object behaves as if open in binary mode. This file should be closed once it is no longer needed.

Modifié dans la version 3.2 : The file object really is a *binary file*; previously it was incorrectly returned in text mode. Also, the *file-like object* now supports the *context manager* protocol : you can use a `with` statement to automatically close it.

> **Note**
>
> Unlike other representations of messages, *file-like* representations are not necessarily independent of the `Mailbox` instance that created them or of the underlying mailbox. More specific documentation is provided by each subclass.

__contains__(*key*)

Return `True` if *key* corresponds to a message, `False` otherwise.

__len__()

Return a count of messages in the mailbox.

clear()

Supprime tous les messages de la boîte de courriel.

pop(*key*, *default=None*)

Return a representation of the message corresponding to *key* and delete the message. If no such message exists, return *default*. The message is represented as an instance of the appropriate format-specific `Message` subclass unless a custom message factory was specified when the `Mailbox` instance was initialized.

popitem()

Return an arbitrary (*key*, *message*) pair, where *key* is a key and *message* is a message representation, and delete the corresponding message. If the mailbox is empty, raise a `KeyError` exception. The message is represented as an instance of the appropriate format-specific `Message` subclass unless a custom message factory was specified when the `Mailbox` instance was initialized.

update(*arg*)

Parameter *arg* should be a *key*-to-*message* mapping or an iterable of (*key*, *message*) pairs. Updates the mailbox so that, for each given *key* and *message*, the message corresponding to *key* is set to *message* as if by using `__setitem__()`. As with `__setitem__()`, each *key* must already correspond to a message in the mailbox or else a `KeyError` exception will be raised, so in general it is incorrect for *arg* to be a `Mailbox` instance.

> **Note**
>
> Unlike with dictionaries, keyword arguments are not supported.

flush()

Write any pending changes to the filesystem. For some `Mailbox` subclasses, changes are always written immediately and `flush()` does nothing, but you should still make a habit of calling this method.

lock()

Acquire an exclusive advisory lock on the mailbox so that other processes know not to modify it. An `ExternalClashError` is raised if the lock is not available. The particular locking mechanisms used depend upon the mailbox format. You should *always* lock the mailbox before making any modifications to its contents.

unlock()

Release the lock on the mailbox, if any.

close()

Flush the mailbox, unlock it if necessary, and close any open files. For some `Mailbox` subclasses, this method does nothing.

`Maildir` objects

class mailbox.**Maildir**(*dirname*, *factory=None*, *create=True*)

A subclass of `Mailbox` for mailboxes in Maildir format. Parameter *factory* is a callable object that accepts a file-like message representation (which behaves as if opened in binary mode) and returns a custom representation. If *factory* is `None`, `MaildirMessage` is used as the default message representation. If *create* is `True`, the mailbox is created if it does not exist.

If *create* is `True` and the *dirname* path exists, it will be treated as an existing maildir without attempting to verify its directory layout.

It is for historical reasons that *dirname* is named as such rather than *path*.

Maildir is a directory-based mailbox format invented for the qmail mail transfer agent and now widely supported by other programs. Messages in a Maildir mailbox are stored in separate files within a common directory structure. This design allows Maildir mailboxes to be accessed and modified by multiple unrelated programs without data corruption, so file locking is unnecessary.

Maildir mailboxes contain three subdirectories, namely : `tmp`, `new`, and `cur`. Messages are created momentarily in the `tmp` subdirectory and then moved to the `new` subdirectory to finalize delivery. A mail user agent may subsequently move the message to the `cur` subdirectory and store information about the state of the message in a special "info" section appended to its file name.

Folders of the style introduced by the Courier mail transfer agent are also supported. Any subdirectory of the main mailbox is considered a folder if `'.'` is the first character in its name. Folder names are represented by `Maildir` without the leading `'.'`. Each folder is itself a Maildir mailbox but should not contain other folders. Instead, a logical nesting is indicated using `'.'` to delimit levels, e.g., "Archived.2005.07".

colon

The Maildir specification requires the use of a colon (`':'`) in certain message file names. However, some operating systems do not permit this character in file names, If you wish to use a Maildir-like format on such an operating system, you should specify another character to use instead. The exclamation point (`'!'`) is a popular choice. For example :

```
import mailbox
mailbox.Maildir.colon = '!'
```

The `colon` attribute may also be set on a per-instance basis.

Modifié dans la version 3.13 : *Maildir* now ignores files with a leading dot.

`Maildir` instances have all of the methods of *Mailbox* in addition to the following :

list_folders()

Return a list of the names of all folders.

get_folder(*folder*)

Return a `Maildir` instance representing the folder whose name is *folder*. A *NoSuchMailboxError* exception is raised if the folder does not exist.

add_folder(*folder*)

Create a folder whose name is *folder* and return a `Maildir` instance representing it.

remove_folder(*folder*)

Delete the folder whose name is *folder*. If the folder contains any messages, a *NotEmptyError* exception will be raised and the folder will not be deleted.

clean()

Delete temporary files from the mailbox that have not been accessed in the last 36 hours. The Maildir specification says that mail-reading programs should do this occasionally.

get_flags(*key*)

Return as a string the flags that are set on the message corresponding to *key*. This is the same as `get_message(key).get_flags()` but much faster, because it does not open the message file. Use this method when iterating over the keys to determine which messages are interesting to get.

If you do have a *MaildirMessage* object, use its *get_flags()* method instead, because changes made by the message's *set_flags()*, *add_flag()* and *remove_flag()* methods are not reflected here until the mailbox's *__setitem__()* method is called.

Ajouté dans la version 3.13.

set_flags(*key*, *flags*)

On the message corresponding to *key*, set the flags specified by *flags* and unset all others. Calling `some_mailbox.set_flags(key, flags)` is similar to

```
one_message = some_mailbox.get_message(key)
one_message.set_flags(flags)
some_mailbox[key] = one_message
```

but faster, because it does not open the message file.

If you do have a `MaildirMessage` object, use its `set_flags()` method instead, because changes made with this mailbox method will not be visible to the message object's method, `get_flags()`.

Ajouté dans la version 3.13.

add_flag(*key*, *flag*)

On the message corresponding to *key*, set the flags specified by *flag* without changing other flags. To add more than one flag at a time, *flag* may be a string of more than one character.

Considerations for using this method versus the message object's `add_flag()` method are similar to those for `set_flags()` ; see the discussion there.

Ajouté dans la version 3.13.

remove_flag(*key*, *flag*)

On the message corresponding to *key*, unset the flags specified by *flag* without changing other flags. To remove more than one flag at a time, *flag* may be a string of more than one character.

Considerations for using this method versus the message object's `remove_flag()` method are similar to those for `set_flags()` ; see the discussion there.

Ajouté dans la version 3.13.

get_info(*key*)

Return a string containing the info for the message corresponding to *key*. This is the same as `get_message(key).get_info()` but much faster, because it does not open the message file. Use this method when iterating over the keys to determine which messages are interesting to get.

If you do have a `MaildirMessage` object, use its `get_info()` method instead, because changes made by the message's `set_info()` method are not reflected here until the mailbox's `__setitem__()` method is called.

Ajouté dans la version 3.13.

set_info(*key*, *info*)

Set the info of the message corresponding to *key* to *info*. Calling `some_mailbox.set_info(key, flags)` is similar to

```
one_message = some_mailbox.get_message(key)
one_message.set_info(info)
some_mailbox[key] = one_message
```

but faster, because it does not open the message file.

If you do have a `MaildirMessage` object, use its `set_info()` method instead, because changes made with this mailbox method will not be visible to the message object's method, `get_info()`.

Ajouté dans la version 3.13.

Some `Mailbox` methods implemented by `Maildir` deserve special remarks :

add(*message*)

__setitem__(*key*, *message*)

update(*arg*)

> **Avertissement**
>
> These methods generate unique file names based upon the current process ID. When using multiple threads, undetected name clashes may occur and cause corruption of the mailbox unless threads are coordinated to avoid using these methods to manipulate the same mailbox simultaneously.

flush()

All changes to Maildir mailboxes are immediately applied, so this method does nothing.

lock()

unlock()

Maildir mailboxes do not support (or require) locking, so these methods do nothing.

close()

`Maildir` instances do not keep any open files and the underlying mailboxes do not support locking, so this method does nothing.

get_file(*key*)

Depending upon the host platform, it may not be possible to modify or remove the underlying message while the returned file remains open.

Voir aussi

maildir man page from Courier
A specification of the format. Describes a common extension for supporting folders.

Using maildir format
Notes on Maildir by its inventor. Includes an updated name-creation scheme and details on "info" semantics.

`mbox` objects

class `mailbox.`**mbox**(*path*, *factory=None*, *create=True*)

A subclass of `Mailbox` for mailboxes in mbox format. Parameter *factory* is a callable object that accepts a file-like message representation (which behaves as if opened in binary mode) and returns a custom representation. If *factory* is `None`, `mboxMessage` is used as the default message representation. If *create* is `True`, the mailbox is created if it does not exist.

The mbox format is the classic format for storing mail on Unix systems. All messages in an mbox mailbox are stored in a single file with the beginning of each message indicated by a line whose first five characters are "From ".

Several variations of the mbox format exist to address perceived shortcomings in the original. In the interest of compatibility, `mbox` implements the original format, which is sometimes referred to as *mboxo*. This means that the `Content-Length` header, if present, is ignored and that any occurrences of "From " at the beginning of a line in a message body are transformed to ">From " when storing the message, although occurrences of ">From " are not transformed to "From " when reading the message.

Some `Mailbox` methods implemented by `mbox` deserve special remarks :

get_file(*key*)

Using the file after calling `flush()` or `close()` on the `mbox` instance may yield unpredictable results or raise an exception.

lock()

unlock()

Three locking mechanisms are used---dot locking and, if available, the `flock()` and `lockf()` system calls.

Voir aussi

mbox man page from tin
A specification of the format, with details on locking.

Configuring Netscape Mail on Unix : Why The Content-Length Format is Bad
An argument for using the original mbox format rather than a variation.

"mbox" is a family of several mutually incompatible mailbox formats
A history of mbox variations.

MH objects

class mailbox.**MH**(*path*, *factory=None*, *create=True*)

A subclass of *Mailbox* for mailboxes in MH format. Parameter *factory* is a callable object that accepts a file-like message representation (which behaves as if opened in binary mode) and returns a custom representation. If *factory* is None, *MHMessage* is used as the default message representation. If *create* is True, the mailbox is created if it does not exist.

MH is a directory-based mailbox format invented for the MH Message Handling System, a mail user agent. Each message in an MH mailbox resides in its own file. An MH mailbox may contain other MH mailboxes (called *folders*) in addition to messages. Folders may be nested indefinitely. MH mailboxes also support *sequences*, which are named lists used to logically group messages without moving them to sub-folders. Sequences are defined in a file called .mh_sequences in each folder.

The MH class manipulates MH mailboxes, but it does not attempt to emulate all of **mh**'s behaviors. In particular, it does not modify and is not affected by the context or .mh_profile files that are used by **mh** to store its state and configuration.

MH instances have all of the methods of *Mailbox* in addition to the following :

Modifié dans la version 3.13 : Supported folders that don't contain a .mh_sequences file.

list_folders()

Return a list of the names of all folders.

get_folder(*folder*)

Return an MH instance representing the folder whose name is *folder*. A *NoSuchMailboxError* exception is raised if the folder does not exist.

add_folder(*folder*)

Create a folder whose name is *folder* and return an MH instance representing it.

remove_folder(*folder*)

Delete the folder whose name is *folder*. If the folder contains any messages, a *NotEmptyError* exception will be raised and the folder will not be deleted.

get_sequences()

Return a dictionary of sequence names mapped to key lists. If there are no sequences, the empty dictionary is returned.

set_sequences(*sequences*)

Re-define the sequences that exist in the mailbox based upon *sequences*, a dictionary of names mapped to key lists, like returned by *get_sequences()*.

pack()

Rename messages in the mailbox as necessary to eliminate gaps in numbering. Entries in the sequences list are updated correspondingly.

> **Note**
>
> Already-issued keys are invalidated by this operation and should not be subsequently used.

Some *Mailbox* methods implemented by MH deserve special remarks :

remove(*key*)

__delitem__(*key*)

discard(*key*)

These methods immediately delete the message. The MH convention of marking a message for deletion by prepending a comma to its name is not used.

lock()

unlock()

Three locking mechanisms are used---dot locking and, if available, the `flock()` and `lockf()` system calls. For MH mailboxes, locking the mailbox means locking the `.mh_sequences` file and, only for the duration of any operations that affect them, locking individual message files.

get_file(*key*)

Depending upon the host platform, it may not be possible to remove the underlying message while the returned file remains open.

flush()

All changes to MH mailboxes are immediately applied, so this method does nothing.

close()

`MH` instances do not keep any open files, so this method is equivalent to `unlock()`.

Voir aussi

nmh - Message Handling System
Home page of **nmh**, an updated version of the original **mh**.

MH & nmh : Email for Users & Programmers
A GPL-licensed book on **mh** and **nmh**, with some information on the mailbox format.

`Babyl` objects

class mailbox.**Babyl**(*path*, *factory=None*, *create=True*)

A subclass of `Mailbox` for mailboxes in Babyl format. Parameter *factory* is a callable object that accepts a file-like message representation (which behaves as if opened in binary mode) and returns a custom representation. If *factory* is `None`, `BabylMessage` is used as the default message representation. If *create* is `True`, the mailbox is created if it does not exist.

Babyl is a single-file mailbox format used by the Rmail mail user agent included with Emacs. The beginning of a message is indicated by a line containing the two characters Control-Underscore (`'\037'`) and Control-L (`'\014'`). The end of a message is indicated by the start of the next message or, in the case of the last message, a line containing a Control-Underscore (`'\037'`) character.

Messages in a Babyl mailbox have two sets of headers, original headers and so-called visible headers. Visible headers are typically a subset of the original headers that have been reformatted or abridged to be more attractive. Each message in a Babyl mailbox also has an accompanying list of *labels*, or short strings that record extra information about the message, and a list of all user-defined labels found in the mailbox is kept in the Babyl options section.

`Babyl` instances have all of the methods of `Mailbox` in addition to the following :

get_labels()

Return a list of the names of all user-defined labels used in the mailbox.

Note

The actual messages are inspected to determine which labels exist in the mailbox rather than consulting the list of labels in the Babyl options section, but the Babyl section is updated whenever the mailbox is modified.

Some `Mailbox` methods implemented by `Babyl` deserve special remarks :

get_file(*key*)

In Babyl mailboxes, the headers of a message are not stored contiguously with the body of the message. To generate a file-like representation, the headers and body are copied together into an `io.BytesIO` instance, which has an API identical to that of a file. As a result, the file-like object is truly independent of the underlying mailbox but does not save memory compared to a string representation.

lock()

unlock()

Three locking mechanisms are used---dot locking and, if available, the `flock()` and `lockf()` system calls.

Voir aussi

Format of Version 5 Babyl Files
A specification of the Babyl format.

Reading Mail with Rmail
The Rmail manual, with some information on Babyl semantics.

`MMDF` objects

class mailbox.**MMDF**(*path*, *factory=None*, *create=True*)

A subclass of `Mailbox` for mailboxes in MMDF format. Parameter *factory* is a callable object that accepts a file-like message representation (which behaves as if opened in binary mode) and returns a custom representation. If *factory* is `None`, `MMDFMessage` is used as the default message representation. If *create* is `True`, the mailbox is created if it does not exist.

MMDF is a single-file mailbox format invented for the Multichannel Memorandum Distribution Facility, a mail transfer agent. Each message is in the same form as an mbox message but is bracketed before and after by lines containing four Control-A (`'\001'`) characters. As with the mbox format, the beginning of each message is indicated by a line whose first five characters are "From ", but additional occurrences of "From " are not transformed to ">From " when storing messages because the extra message separator lines prevent mistaking such occurrences for the starts of subsequent messages.

Some `Mailbox` methods implemented by `MMDF` deserve special remarks :

get_file(*key*)

Using the file after calling `flush()` or `close()` on the `MMDF` instance may yield unpredictable results or raise an exception.

lock()

unlock()

Three locking mechanisms are used---dot locking and, if available, the `flock()` and `lockf()` system calls.

Voir aussi

mmdf man page from tin
A specification of MMDF format from the documentation of tin, a newsreader.

MMDF
A Wikipedia article describing the Multichannel Memorandum Distribution Facility.

20.3.2 `Message` objects

class mailbox.**Message**(*message=None*)

A subclass of the `email.message` module's `Message`. Subclasses of `mailbox.Message` add mailbox-format-specific state and behavior.

If *message* is omitted, the new instance is created in a default, empty state. If *message* is an `email.message.Message` instance, its contents are copied ; furthermore, any format-specific information is converted insofar as possible if *message* is a `Message` instance. If *message* is a string, a byte string, or a file, it should contain an **RFC 2822**-compliant message, which is read and parsed. Files should be open in binary mode, but text mode files are accepted for backward compatibility.

The format-specific state and behaviors offered by subclasses vary, but in general it is only the properties that are not specific to a particular mailbox that are supported (although presumably the properties are specific to a particular mailbox format). For example, file offsets for single-file mailbox formats and file names for directory-based mailbox formats are not retained, because they are only applicable to the original mailbox. But state such as whether a message has been read by the user or marked as important is retained, because it applies to the message itself.

There is no requirement that `Message` instances be used to represent messages retrieved using *`Mailbox`* instances. In some situations, the time and memory required to generate `Message` representations might not be acceptable. For such situations, `Mailbox` instances also offer string and file-like representations, and a custom message factory may be specified when a `Mailbox` instance is initialized.

`MaildirMessage` objects

class `mailbox.`**`MaildirMessage`**(*message=None*)

A message with Maildir-specific behaviors. Parameter *message* has the same meaning as with the *`Message`* constructor.

Typically, a mail user agent application moves all of the messages in the `new` subdirectory to the `cur` subdirectory after the first time the user opens and closes the mailbox, recording that the messages are old whether or not they've actually been read. Each message in `cur` has an "info" section added to its file name to store information about its state. (Some mail readers may also add an "info" section to messages in `new`.) The "info" section may take one of two forms : it may contain "2," followed by a list of standardized flags (e.g., "2,FR") or it may contain "1," followed by so-called experimental information. Standard flags for Maildir messages are as follows :

Option	Signification	Explication
D	Draft	Under composition
F	Flagged	Marked as important
P	Passed	Forwarded, resent, or bounced
R	Replied	Replied to
S	Seen	Read
T	Trashed	Marked for subsequent deletion

`MaildirMessage` instances offer the following methods :

`get_subdir`()

Return either "new" (if the message should be stored in the `new` subdirectory) or "cur" (if the message should be stored in the `cur` subdirectory).

> **Note**
>
> A message is typically moved from `new` to `cur` after its mailbox has been accessed, whether or not the message has been read. A message `msg` has been read if `"S" in msg.get_flags()` is `True`.

`set_subdir`(*subdir*)

Set the subdirectory the message should be stored in. Parameter *subdir* must be either "new" or "cur".

`get_flags`()

Return a string specifying the flags that are currently set. If the message complies with the standard Maildir format, the result is the concatenation in alphabetical order of zero or one occurrence of each of `'D'`, `'F'`, `'P'`, `'R'`, `'S'`, and `'T'`. The empty string is returned if no flags are set or if "info" contains experimental semantics.

`set_flags`(*flags*)

Set the flags specified by *flags* and unset all others.

add_flag(*flag*)

Set the flag(s) specified by *flag* without changing other flags. To add more than one flag at a time, *flag* may be a string of more than one character. The current "info" is overwritten whether or not it contains experimental information rather than flags.

remove_flag(*flag*)

Unset the flag(s) specified by *flag* without changing other flags. To remove more than one flag at a time, *flag* maybe a string of more than one character. If "info" contains experimental information rather than flags, the current "info" is not modified.

get_date()

Return the delivery date of the message as a floating-point number representing seconds since the epoch.

set_date(*date*)

Set the delivery date of the message to *date*, a floating-point number representing seconds since the epoch.

get_info()

Return a string containing the "info" for a message. This is useful for accessing and modifying "info" that is experimental (i.e., not a list of flags).

set_info(*info*)

Set "info" to *info*, which should be a string.

When a `MaildirMessage` instance is created based upon an `mboxMessage` or `MMDFMessage` instance, the `Status` and `X-Status` headers are omitted and the following conversions take place :

Resulting state	`mboxMessage` or `MMDFMessage` state
"cur" subdirectory	O flag
F flag	F flag
R flag	A flag
S flag	R flag
T flag	D flag

When a `MaildirMessage` instance is created based upon an `MHMessage` instance, the following conversions take place :

Resulting state	`MHMessage` state
"cur" subdirectory	"unseen" sequence
"cur" subdirectory and S flag	no "unseen" sequence
F flag	"flagged" sequence
R flag	"replied" sequence

When a `MaildirMessage` instance is created based upon a `BabylMessage` instance, the following conversions take place :

Resulting state	`BabylMessage` state
"cur" subdirectory	"unseen" label
"cur" subdirectory and S flag	no "unseen" label
P flag	"forwarded" or "resent" label
R flag	"answered" label
T flag	"deleted" label

`mboxMessage` objects

class `mailbox.`**`mboxMessage`**(*message=None*)

A message with mbox-specific behaviors. Parameter *message* has the same meaning as with the `Message` constructor.

Messages in an mbox mailbox are stored together in a single file. The sender's envelope address and the time of delivery are typically stored in a line beginning with "From " that is used to indicate the start of a message, though there is considerable variation in the exact format of this data among mbox implementations. Flags that indicate the state of the message, such as whether it has been read or marked as important, are typically stored in `Status` and `X-Status` headers.

Conventional flags for mbox messages are as follows :

Option	Signification	Explication
R	Read	Read
O	Old	Previously detected by MUA
D	Deleted	Marked for subsequent deletion
F	Flagged	Marked as important
A	Answered	Replied to

The "R" and "O" flags are stored in the `Status` header, and the "D", "F", and "A" flags are stored in the `X-Status` header. The flags and headers typically appear in the order mentioned.

`mboxMessage` instances offer the following methods :

`get_from`()

Return a string representing the "From " line that marks the start of the message in an mbox mailbox. The leading "From " and the trailing newline are excluded.

`set_from`(*from_*, *time_=None*)

Set the "From " line to *from_*, which should be specified without a leading "From " or trailing newline. For convenience, *time_* may be specified and will be formatted appropriately and appended to *from_*. If *time_* is specified, it should be a `time.struct_time` instance, a tuple suitable for passing to `time.strftime()`, or `True` (to use `time.gmtime()`).

`get_flags`()

Return a string specifying the flags that are currently set. If the message complies with the conventional format, the result is the concatenation in the following order of zero or one occurrence of each of `'R'`, `'O'`, `'D'`, `'F'`, and `'A'`.

`set_flags`(*flags*)

Set the flags specified by *flags* and unset all others. Parameter *flags* should be the concatenation in any order of zero or more occurrences of each of `'R'`, `'O'`, `'D'`, `'F'`, and `'A'`.

`add_flag`(*flag*)

Set the flag(s) specified by *flag* without changing other flags. To add more than one flag at a time, *flag* may be a string of more than one character.

`remove_flag`(*flag*)

Unset the flag(s) specified by *flag* without changing other flags. To remove more than one flag at a time, *flag* maybe a string of more than one character.

When an `mboxMessage` instance is created based upon a `MaildirMessage` instance, a "From " line is generated based upon the `MaildirMessage` instance's delivery date, and the following conversions take place :

Resulting state	`MaildirMessage` state
R flag	S flag
O flag	”cur” subdirectory
D flag	T flag
F flag	F flag
A flag	R flag

When an `mboxMessage` instance is created based upon an `MHMessage` instance, the following conversions take place :

Resulting state	`MHMessage` state
R flag and O flag	no ”unseen” sequence
O flag	”unseen” sequence
F flag	”flagged” sequence
A flag	”replied” sequence

When an `mboxMessage` instance is created based upon a `BabylMessage` instance, the following conversions take place :

Resulting state	`BabylMessage` state
R flag and O flag	no ”unseen” label
O flag	”unseen” label
D flag	”deleted” label
A flag	”answered” label

When a `mboxMessage` instance is created based upon an `MMDFMessage` instance, the ”From ” line is copied and all flags directly correspond :

Resulting state	`MMDFMessage` state
R flag	R flag
O flag	O flag
D flag	D flag
F flag	F flag
A flag	A flag

`MHMessage` objects

class `mailbox.`**`MHMessage`**(*message=None*)

A message with MH-specific behaviors. Parameter *message* has the same meaning as with the `Message` constructor.

MH messages do not support marks or flags in the traditional sense, but they do support sequences, which are logical groupings of arbitrary messages. Some mail reading programs (although not the standard **mh** and **nmh**) use sequences in much the same way flags are used with other formats, as follows :

Séquence	Explication
unseen	Not read, but previously detected by MUA
replied	Replied to
flagged	Marked as important

`MHMessage` instances offer the following methods :

get_sequences()

Return a list of the names of sequences that include this message.

set_sequences(*sequences*)

Set the list of sequences that include this message.

add_sequence(*sequence*)

Add *sequence* to the list of sequences that include this message.

remove_sequence(*sequence*)

Remove *sequence* from the list of sequences that include this message.

When an `MHMessage` instance is created based upon a `MaildirMessage` instance, the following conversions take place :

Resulting state	`MaildirMessage` state
"unseen" sequence	no S flag
"replied" sequence	R flag
"flagged" sequence	F flag

When an `MHMessage` instance is created based upon an `mboxMessage` or `MMDFMessage` instance, the `Status` and `X-Status` headers are omitted and the following conversions take place :

Resulting state	`mboxMessage` or `MMDFMessage` state
"unseen" sequence	no R flag
"replied" sequence	A flag
"flagged" sequence	F flag

When an `MHMessage` instance is created based upon a `BabylMessage` instance, the following conversions take place :

Resulting state	`BabylMessage` state
"unseen" sequence	"unseen" label
"replied" sequence	"answered" label

`BabylMessage` objects

class mailbox.**BabylMessage**(*message=None*)

A message with Babyl-specific behaviors. Parameter *message* has the same meaning as with the `Message` constructor.

Certain message labels, called *attributes*, are defined by convention to have special meanings. The attributes are as follows :

Label	Explication
unseen	Not read, but previously detected by MUA
deleted	Marked for subsequent deletion
filed	Copied to another file or mailbox
answered	Replied to
forwarded	Forwarded
edited	Modified by the user
resent	Resent

By default, Rmail displays only visible headers. The `BabylMessage` class, though, uses the original headers because they are more complete. Visible headers may be accessed explicitly if desired.

`BabylMessage` instances offer the following methods :

get_labels()

Return a list of labels on the message.

set_labels(*labels*)

Set the list of labels on the message to *labels*.

add_label(*label*)

Add *label* to the list of labels on the message.

remove_label(*label*)

Remove *label* from the list of labels on the message.

get_visible()

Return a `Message` instance whose headers are the message's visible headers and whose body is empty.

set_visible(*visible*)

Set the message's visible headers to be the same as the headers in *message*. Parameter *visible* should be a `Message` instance, an `email.message.Message` instance, a string, or a file-like object (which should be open in text mode).

update_visible()

When a `BabylMessage` instance's original headers are modified, the visible headers are not automatically modified to correspond. This method updates the visible headers as follows : each visible header with a corresponding original header is set to the value of the original header, each visible header without a corresponding original header is removed, and any of `Date`, `From`, `Reply-To`, `To`, `CC`, and `Subject` that are present in the original headers but not the visible headers are added to the visible headers.

When a `BabylMessage` instance is created based upon a `MaildirMessage` instance, the following conversions take place :

Resulting state	`MaildirMessage` state
"unseen" label	no S flag
"deleted" label	T flag
"answered" label	R flag
"forwarded" label	P flag

When a `BabylMessage` instance is created based upon an `mboxMessage` or `MMDFMessage` instance, the `Status` and `X-Status` headers are omitted and the following conversions take place :

Resulting state	`mboxMessage` or `MMDFMessage` state
"unseen" label	no R flag
"deleted" label	D flag
"answered" label	A flag

When a `BabylMessage` instance is created based upon an `MHMessage` instance, the following conversions take place :

Resulting state	`MHMessage` state
"unseen" label	"unseen" sequence
"answered" label	"replied" sequence

MMDFMessage objects

class mailbox.**MMDFMessage**(*message=None*)

Un message avec des comportements spécifiques à *MMDF*. Le paramètre *message* a le même sens que pour le constructeur de `Message`.

Comme pour le message d'une boîte de courriel *mbox*, les messages *MMDF* sont stockés avec l'adresse de l'expéditeur et la date d'expédition dans la ligne initiale commençant avec « From ». De même, les options indiquant l'état du message sont stockées dans les en-têtes `Status` et `X-Status`.

Les options conventionnelles des messages *MMDF* sont identiques à celles de message *mbox* et sont les suivantes :

Option	Signification	Explication
R	Read	Read
O	Old	Previously detected by MUA
D	Deleted	Marked for subsequent deletion
F	Flagged	Marked as important
A	Answered	Replied to

The "R" and "O" flags are stored in the `Status` header, and the "D", "F", and "A" flags are stored in the `X-Status` header. The flags and headers typically appear in the order mentioned.

`MMDFMessage` instances offer the following methods, which are identical to those offered by `mboxMessage` :

get_from()

Return a string representing the "From " line that marks the start of the message in an mbox mailbox. The leading "From " and the trailing newline are excluded.

set_from(*from_*, *time_=None*)

Set the "From " line to *from_*, which should be specified without a leading "From " or trailing newline. For convenience, *time_* may be specified and will be formatted appropriately and appended to *from_*. If *time_* is specified, it should be a `time.struct_time` instance, a tuple suitable for passing to `time.strftime()`, or `True` (to use `time.gmtime()`).

get_flags()

Return a string specifying the flags that are currently set. If the message complies with the conventional format, the result is the concatenation in the following order of zero or one occurrence of each of `'R'`, `'O'`, `'D'`, `'F'`, and `'A'`.

set_flags(*flags*)

Set the flags specified by *flags* and unset all others. Parameter *flags* should be the concatenation in any order of zero or more occurrences of each of `'R'`, `'O'`, `'D'`, `'F'`, and `'A'`.

add_flag(*flag*)

Set the flag(s) specified by *flag* without changing other flags. To add more than one flag at a time, *flag* may be a string of more than one character.

remove_flag(*flag*)

Unset the flag(s) specified by *flag* without changing other flags. To remove more than one flag at a time, *flag* maybe a string of more than one character.

When an `MMDFMessage` instance is created based upon a `MaildirMessage` instance, a "From " line is generated based upon the `MaildirMessage` instance's delivery date, and the following conversions take place :

Resulting state	`MaildirMessage` state
R flag	S flag
O flag	”cur” subdirectory
D flag	T flag
F flag	F flag
A flag	R flag

When an `MMDFMessage` instance is created based upon an `MHMessage` instance, the following conversions take place :

Resulting state	`MHMessage` state
R flag and O flag	no ”unseen” sequence
O flag	”unseen” sequence
F flag	”flagged” sequence
A flag	”replied” sequence

When an `MMDFMessage` instance is created based upon a `BabylMessage` instance, the following conversions take place :

Resulting state	`BabylMessage` state
R flag and O flag	no ”unseen” label
O flag	”unseen” label
D flag	”deleted” label
A flag	”answered” label

When an `MMDFMessage` instance is created based upon an `mboxMessage` instance, the ”From ” line is copied and all flags directly correspond :

Resulting state	état de `mboxMessage`
R flag	R flag
O flag	O flag
D flag	D flag
F flag	F flag
A flag	A flag

20.3.3 Exceptions

The following exception classes are defined in the `mailbox` module :

exception `mailbox.`**`Error`**

Classe de base pour toutes les autres exceptions spécifiques à ce module.

exception `mailbox.`**`NoSuchMailboxError`**

Levée lorsqu'une boîte de courriel est attendue mais introuvable, comme quand on instancie une sous-classe `Mailbox` avec un chemin qui n'existe pas (et avec le paramètre *create* fixé à `False`), ou quand on ouvre un répertoire qui n'existe pas.

exception `mailbox.`**`NotEmptyError`**

Levée lorsqu'une boîte de courriel n'est pas vide mais devrait l'être, comme lorsqu'on supprime un répertoire contenant des messages.

exception mailbox.**ExternalClashError**

Raised when some mailbox-related condition beyond the control of the program causes it to be unable to proceed, such as when failing to acquire a lock that another program already holds a lock, or when a uniquely generated file name already exists.

exception mailbox.**FormatError**

Levée lorsque la donnée dans le fichier ne peut être analysée, comme lorsque l'instance de *MH* tente de lire un fichier `.mh_sequences` corrompu.

20.3.4 Exemples

Un exemple simple d'affichage de l'objet, qui semble pertinent, de tous les messages d'une boîte de courriel :

```
import mailbox
for message in mailbox.mbox('~/mbox'):
    subject = message['subject']       # Could possibly be None.
    if subject and 'python' in subject.lower():
        print(subject)
```

Cet exemple copie tout le courriel d'une boite de courriel au format *Babyl* vers une boite de courriel au format *MH*, convertissant toute l'information qu'il est possible de convertir du premier format vers le second :

```
import mailbox
destination = mailbox.MH('~/Mail')
destination.lock()
for message in mailbox.Babyl('~/RMAIL'):
    destination.add(mailbox.MHMessage(message))
destination.flush()
destination.unlock()
```

Cet exemple trie le courriel en provenance de plusieurs listes de diffusion vers différentes boîtes de courriel, tout en évitant une corruption à cause de modifications concurrentielles par d'autres programmes, une perte due à une interruption du programme ou un arrêt prématuré causé par des messages mal structurés :

```
import mailbox
import email.errors

list_names = ('python-list', 'python-dev', 'python-bugs')

boxes = {name: mailbox.mbox('~/email/%s' % name) for name in list_names}
inbox = mailbox.Maildir('~/Maildir', factory=None)

for key in inbox.iterkeys():
    try:
        message = inbox[key]
    except email.errors.MessageParseError:
        continue                # The message is malformed. Just leave it.

    for name in list_names:
        list_id = message['list-id']
        if list_id and name in list_id:
            # Get mailbox to use
            box = boxes[name]

            # Write copy to disk before removing original.
            # If there's a crash, you might duplicate a message, but
            # that's better than losing a message completely.
            box.lock()
```

(suite sur la page suivante)

(suite de la page précédente)

```
            box.add(message)
            box.flush()
            box.unlock()

            # Remove original message
            inbox.lock()
            inbox.discard(key)
            inbox.flush()
            inbox.unlock()
            break               # Found destination, so stop looking.

for box in boxes.itervalues():
    box.close()
```

20.4 mimetypes --- Map filenames to MIME types

Source code : Lib/mimetypes.py

The *mimetypes* module converts between a filename or URL and the MIME type associated with the filename extension. Conversions are provided from filename to MIME type and from MIME type to filename extension ; encodings are not supported for the latter conversion.

The module provides one class and a number of convenience functions. The functions are the normal interface to this module, but some applications may be interested in the class as well.

The functions described below provide the primary interface for this module. If the module has not been initialized, they will call *init()* if they rely on the information *init()* sets up.

mimetypes.**guess_type**(*url*, *strict=True*)

Guess the type of a file based on its filename, path or URL, given by *url*. URL can be a string or a *path-like object*.

The return value is a tuple `(type, encoding)` where *type* is `None` if the type can't be guessed (missing or unknown suffix) or a string of the form `'type/subtype'`, usable for a MIME *content-type* header.

encoding is `None` for no encoding or the name of the program used to encode (e.g. **compress** or **gzip**). The encoding is suitable for use as a *Content-Encoding* header, **not** as a *Content-Transfer-Encoding* header. The mappings are table driven. Encoding suffixes are case sensitive ; type suffixes are first tried case sensitively, then case insensitively.

The optional *strict* argument is a flag specifying whether the list of known MIME types is limited to only the official types registered with IANA. However, the behavior of this module also depends on the underlying operating system. Only file types recognized by the OS or explicitly registered with Python's internal database can be identified. When *strict* is `True` (the default), only the IANA types are supported ; when *strict* is `False`, some additional non-standard but commonly used MIME types are also recognized.

Modifié dans la version 3.8 : Added support for *url* being a *path-like object*.

Obsolète depuis la version 3.13 : Passing a file path instead of URL is *soft deprecated*. Use *guess_file_type()* for this.

mimetypes.**guess_file_type**(*path*, *, *strict=True*)

Guess the type of a file based on its path, given by *path*. Similar to the *guess_type()* function, but accepts a path instead of URL. Path can be a string, a bytes object or a *path-like object*.

Ajouté dans la version 3.13.

mimetypes.**guess_all_extensions**(*type*, *strict=True*)

Guess the extensions for a file based on its MIME type, given by *type*. The return value is a list of strings

giving all possible filename extensions, including the leading dot (`'.'`). The extensions are not guaranteed to have been associated with any particular data stream, but would be mapped to the MIME type *type* by `guess_type()` and `guess_file_type()`.

The optional *strict* argument has the same meaning as with the `guess_type()` function.

`mimetypes.`**`guess_extension`**(*type*, *strict=True*)

Guess the extension for a file based on its MIME type, given by *type*. The return value is a string giving a filename extension, including the leading dot (`'.'`). The extension is not guaranteed to have been associated with any particular data stream, but would be mapped to the MIME type *type* by `guess_type()` and `guess_file_type()`. If no extension can be guessed for *type*, `None` is returned.

The optional *strict* argument has the same meaning as with the `guess_type()` function.

Some additional functions and data items are available for controlling the behavior of the module.

`mimetypes.`**`init`**(*files=None*)

Initialize the internal data structures. If given, *files* must be a sequence of file names which should be used to augment the default type map. If omitted, the file names to use are taken from `knownfiles`; on Windows, the current registry settings are loaded. Each file named in *files* or `knownfiles` takes precedence over those named before it. Calling `init()` repeatedly is allowed.

Specifying an empty list for *files* will prevent the system defaults from being applied : only the well-known values will be present from a built-in list.

If *files* is `None` the internal data structure is completely rebuilt to its initial default value. This is a stable operation and will produce the same results when called multiple times.

Modifié dans la version 3.2 : Previously, Windows registry settings were ignored.

`mimetypes.`**`read_mime_types`**(*filename*)

Load the type map given in the file *filename*, if it exists. The type map is returned as a dictionary mapping filename extensions, including the leading dot (`'.'`), to strings of the form `'type/subtype'`. If the file *filename* does not exist or cannot be read, `None` is returned.

`mimetypes.`**`add_type`**(*type*, *ext*, *strict=True*)

Add a mapping from the MIME type *type* to the extension *ext*. When the extension is already known, the new type will replace the old one. When the type is already known the extension will be added to the list of known extensions.

When *strict* is `True` (the default), the mapping will be added to the official MIME types, otherwise to the non-standard ones.

`mimetypes.`**`inited`**

Flag indicating whether or not the global data structures have been initialized. This is set to `True` by `init()`.

`mimetypes.`**`knownfiles`**

List of type map file names commonly installed. These files are typically named `mime.types` and are installed in different locations by different packages.

`mimetypes.`**`suffix_map`**

Dictionary mapping suffixes to suffixes. This is used to allow recognition of encoded files for which the encoding and the type are indicated by the same extension. For example, the `.tgz` extension is mapped to `.tar.gz` to allow the encoding and type to be recognized separately.

`mimetypes.`**`encodings_map`**

Dictionary mapping filename extensions to encoding types.

`mimetypes.`**`types_map`**

Dictionary mapping filename extensions to MIME types.

`mimetypes.`**`common_types`**

Dictionary mapping filename extensions to non-standard, but commonly found MIME types.

Un exemple d'utilisation du module :

```
>>> import mimetypes
>>> mimetypes.init()
>>> mimetypes.knownfiles
['/etc/mime.types', '/etc/httpd/mime.types', ... ]
>>> mimetypes.suffix_map['.tgz']
'.tar.gz'
>>> mimetypes.encodings_map['.gz']
'gzip'
>>> mimetypes.types_map['.tgz']
'application/x-tar-gz'
```

20.4.1 MimeTypes Objects

The `MimeTypes` class may be useful for applications which may want more than one MIME-type database ; it provides an interface similar to the one of the `mimetypes` module.

class mimetypes.**MimeTypes**(*filenames=()*, *strict=True*)

This class represents a MIME-types database. By default, it provides access to the same database as the rest of this module. The initial database is a copy of that provided by the module, and may be extended by loading additional `mime.types`-style files into the database using the `read()` or `readfp()` methods. The mapping dictionaries may also be cleared before loading additional data if the default data is not desired.

The optional *filenames* parameter can be used to cause additional files to be loaded "on top" of the default database.

suffix_map

Dictionary mapping suffixes to suffixes. This is used to allow recognition of encoded files for which the encoding and the type are indicated by the same extension. For example, the `.tgz` extension is mapped to `.tar.gz` to allow the encoding and type to be recognized separately. This is initially a copy of the global `suffix_map` defined in the module.

encodings_map

Dictionary mapping filename extensions to encoding types. This is initially a copy of the global `encodings_map` defined in the module.

types_map

Tuple containing two dictionaries, mapping filename extensions to MIME types : the first dictionary is for the non-standards types and the second one is for the standard types. They are initialized by `common_types` and `types_map`.

types_map_inv

Tuple containing two dictionaries, mapping MIME types to a list of filename extensions : the first dictionary is for the non-standards types and the second one is for the standard types. They are initialized by `common_types` and `types_map`.

guess_extension(*type*, *strict=True*)

Similar to the `guess_extension()` function, using the tables stored as part of the object.

guess_type(*url*, *strict=True*)

Similar to the `guess_type()` function, using the tables stored as part of the object.

guess_file_type(*path*, ***, *strict=True*)

Similar to the `guess_file_type()` function, using the tables stored as part of the object.

Ajouté dans la version 3.13.

guess_all_extensions(*type*, *strict=True*)

Similar to the `guess_all_extensions()` function, using the tables stored as part of the object.

read(*filename*, *strict=True*)

Load MIME information from a file named *filename*. This uses `readfp()` to parse the file.

If *strict* is `True`, information will be added to list of standard types, else to the list of non-standard types.

readfp(*fp*, *strict=True*)

Load MIME type information from an open file *fp*. The file must have the format of the standard `mime.types` files.

If *strict* is `True`, information will be added to the list of standard types, else to the list of non-standard types.

read_windows_registry(*strict=True*)

Load MIME type information from the Windows registry.

Availability : Windows.

If *strict* is `True`, information will be added to the list of standard types, else to the list of non-standard types.

Ajouté dans la version 3.2.

add_type(*type*, *ext*, *strict=True*)

Add a mapping from the MIME type *type* to the extension *ext*. When the extension is already known, the new type will replace the old one. When the type is already known the extension will be added to the list of known extensions.

When *strict* is `True` (the default), the mapping will be added to the official MIME types, otherwise to the non-standard ones.

20.5 `base64` --- Base16, Base32, Base64, Base85 Data Encodings

Code source : Lib/base64.py

This module provides functions for encoding binary data to printable ASCII characters and decoding such encodings back to binary data. It provides encoding and decoding functions for the encodings specified in **RFC 4648**, which defines the Base16, Base32, and Base64 algorithms, and for the de-facto standard Ascii85 and Base85 encodings.

The **RFC 4648** encodings are suitable for encoding binary data so that it can be safely sent by email, used as parts of URLs, or included as part of an HTTP POST request. The encoding algorithm is not the same as the **uuencode** program.

There are two interfaces provided by this module. The modern interface supports encoding *bytes-like objects* to ASCII `bytes`, and decoding *bytes-like objects* or strings containing ASCII to `bytes`. Both base-64 alphabets defined in **RFC 4648** (normal, and URL- and filesystem-safe) are supported.

L'interface historique ne permet pas le décodage des chaînes de caractères mais fournit des fonctions permettant d'encoder et décoder depuis et vers des *objets fichiers*. Elle ne gère que l'alphabet base64 standard et ajoute une nouvelle ligne tous les 76 caractères, comme spécifié par la **RFC 2045**. Notez que le paquet `email` est probablement ce que vous cherchez si vous souhaitez une implémentation de la **RFC 2045**.

Modifié dans la version 3.3 : Les chaînes de caractères Unicode contenant uniquement des caractères ASCII sont désormais acceptées par les fonctions de décodage de l'interface moderne.

Modifié dans la version 3.4 : Tous les *objets octet-compatibles* sont désormais acceptés par l'ensemble des fonctions d'encodage et de décodage de ce module. La gestion de Ascii85/base85 a été ajoutée.

L'interface moderne propose :

`base64`.**b64encode**(*s*, *altchars=None*)

Encode un *objet octet-compatible* *s* en utilisant l'algorithme base64 et renvoie les `bytes` encodés.

Optional *altchars* must be a *bytes-like object* of length 2 which specifies an alternative alphabet for the `+` and `/` characters. This allows an application to e.g. generate URL or filesystem safe Base64 strings. The default is `None`, for which the standard Base64 alphabet is used.

May assert or raise a `ValueError` if the length of *altchars* is not 2. Raises a `TypeError` if *altchars* is not a *bytes-like object*.

`base64.b64decode`(*s*, *altchars=None*, *validate=False*)

Décode un *objet octet-compatible* ou une chaîne de caractères ASCII *s* encodée en base64 et renvoie les `bytes` décodés.

Optional *altchars* must be a *bytes-like object* or ASCII string of length 2 which specifies the alternative alphabet used instead of the `+` and `/` characters.

Une exception `binascii.Error` est levée si *s* n'est pas remplie à une longueur attendue.

Si *validate* est `False` (par défaut), les caractères qui ne sont ni dans l'alphabet base64 normal, ni dans l'alphabet alternatif, sont ignorés avant la vérification de la longueur du remplissage. Si *validate* est `True`, les caractères hors de l'alphabet de l'entrée produisent une `binascii.Error`.

For more information about the strict base64 check, see `binascii.a2b_base64()`

May assert or raise a `ValueError` if the length of *altchars* is not 2.

`base64.standard_b64encode`(*s*)

Encode un *objet octet-compatible* *s* en utilisant l'alphabet standard base64 et renvoie les `bytes` encodés.

`base64.standard_b64decode`(*s*)

Décode un *objet octet-compatible* ou une chaîne de caractères ASCII *s* utilisant l'alphabet base64 standard et renvoie les `bytes` décodés.

`base64.urlsafe_b64encode`(*s*)

Encode un *objet byte-compatible* *s* en utilisant un alphabet sûr pour les URL et systèmes de fichiers qui substitue `-` et `_` à `+` et `/` dans l'alphabet standard base64 et renvoie les `bytes` encodés.

`base64.urlsafe_b64decode`(*s*)

Décode un *objet octet-compatible* ou une chaîne de caractères ASCII *s* utilisant un alphabet sûr pour les URL et systèmes de fichiers qui substitue `-` et `_` à `+` et `/` dans l'alphabet standard base64 et renvoie les `bytes` décodés.

`base64.b32encode`(*s*)

Encode un *objet byte-compatible* *s* en utilisant l'algorithme base32 et renvoie les `bytes` encodés.

`base64.b32decode`(*s*, *casefold=False*, *map01=None*)

Décode un *objet octet-compatible* ou une chaîne de caractères ASCII *s* encodé en base32 et renvoie les `bytes` décodés.

L'option *casefold* est un drapeau spécifiant si l'utilisation d'un alphabet en minuscules est acceptable comme entrée. Pour des raisons de sécurité, cette option est à `False` par défaut.

RFC 4648 allows for optional mapping of the digit 0 (zero) to the letter O (oh), and for optional mapping of the digit 1 (one) to either the letter I (eye) or letter L (el). The optional argument *map01* when not `None`, specifies which letter the digit 1 should be mapped to (when *map01* is not `None`, the digit 0 is always mapped to the letter O). For security purposes the default is `None`, so that 0 and 1 are not allowed in the input.

Une exception `binascii.Error` est levée si *s* n'est pas remplie à une longueur attendue ou si elle contient des caractères hors de l'alphabet.

`base64.b32hexencode`(*s*)

Similar to `b32encode()` but uses the Extended Hex Alphabet, as defined in **RFC 4648**.

Ajouté dans la version 3.10.

`base64.b32hexdecode`(*s*, *casefold=False*)

Similar to `b32decode()` but uses the Extended Hex Alphabet, as defined in **RFC 4648**.

This version does not allow the digit 0 (zero) to the letter O (oh) and digit 1 (one) to either the letter I (eye) or letter L (el) mappings, all these characters are included in the Extended Hex Alphabet and are not interchangeable.

Ajouté dans la version 3.10.

base64.**b16encode**(*s*)

Encode un *objet byte-compatible* *s* en utilisant l'algorithme base16 et renvoie les `bytes` encodés.

base64.**b16decode**(*s*, *casefold=False*)

Décode un *objet octet-compatible* ou une chaîne de caractères ASCII *s* encodé en base16 et renvoie les `bytes` décodés.

L'option *casefold* est un drapeau spécifiant si l'utilisation d'un alphabet en minuscules est acceptable comme entrée. Pour des raisons de sécurité, cette option est à `False` par défaut.

Une exception `binascii.Error` est levée si *s* n'est pas remplie à une longueur attendue ou si elle contient des caractères hors de l'alphabet.

base64.**a85encode**(*b*, **, foldspaces=False*, *wrapcol=0*, *pad=False*, *adobe=False*)

Encode un *objet byte-compatible* *s* en utilisant l'algorithme Ascii85 et renvoie les `bytes` encodés.

L'option *foldspaces* permet d'utiliser la séquence spéciale 'y' à la place de quatre espaces consécutives (ASCII `0x20`) comme pris en charge par *btoa*. Cette fonctionnalité n'est pas gérée par l'encodage « standard » Ascii85.

wrapcol controls whether the output should have newline (`b'\n'`) characters added to it. If this is non-zero, each output line will be at most this many characters long, excluding the trailing newline.

pad spécifie l'ajout de caractères de remplissage (*padding* en anglais) à l'entrée jusqu'à ce que sa longueur soit un multiple de 4 avant encodage. Notez que l'implémentation `btoa` effectue systématiquement ce remplissage.

adobe contrôle si oui ou non la séquence encodée d'octets est encadrée par `<~` et `~>` comme utilisé dans l'implémentation Adobe.

Ajouté dans la version 3.4.

base64.**a85decode**(*b*, **, foldspaces=False*, *adobe=False*, *ignorechars=b' \t\n\r\x0b'*)

Décode un *objet octet-compatible* ou une chaîne de caractères ASCII *s* encodé en Ascii85 et renvoie les `bytes` décodés.

L'option *foldspaces* permet d'utiliser la séquence spéciale 'y' à la place de quatre espaces consécutives (ASCII `0x20`) comme pris en charge par *btoa*. Cette fonctionnalité n'est pas gérée par l'encodage « standard » Ascii85.

adobe indique si la séquence d'entrée utilise le format Adobe Ascii85 (c'est-à-dire utilise l'encadrement par `<~` et `~>`).

ignorechars doit être un *bytes-like object* ou une chaîne ASCII contenant des caractères à ignorer dans l'entrée. Il ne doit contenir que des caractères d'espacement et contient par défaut l'ensemble des caractères d'espacement de l'alphabet ASCII.

Ajouté dans la version 3.4.

base64.**b85encode**(*b*, *pad=False*)

Encode un *objet byte-compatible* *s* en utilisant l'algorithme base85 (tel qu'utilisé par exemple par le programme *git-diff* sur des données binaires) et renvoie les `bytes` encodés.

Si *pad* est vrai, des caractères de remplissage `b'\0'` (*padding* en anglais) sont ajoutés à l'entrée jusqu'à ce que sa longueur soit un multiple de 4 octets avant encodage.

Ajouté dans la version 3.4.

base64.**b85decode**(*b*)

Décode un *objet octet-compatible* ou une chaîne de caractères ASCII *b* encodé en base85 et renvoie les `bytes` décodés. Les caractères de remplissage sont implicitement retirés si nécessaire.

Ajouté dans la version 3.4.

base64.**z85encode**(*s*)

Encode the *bytes-like object* *s* using Z85 (as used in ZeroMQ) and return the encoded `bytes`. See Z85 specification for more information.

Ajouté dans la version 3.13.

`base64.z85decode(s)`

Decode the Z85-encoded *bytes-like object* or ASCII string *s* and return the decoded `bytes`. See Z85 specification for more information.

Ajouté dans la version 3.13.

L'interface historique :

`base64.decode(input, output)`

Décode le contenu d'un fichier binaire *input* et écrit les données binaires résultantes dans le fichier *output*. *input* et *output* doivent être des *objets fichiers*. *input* est lu jusqu'à ce que `input.readline()` renvoie un objet *bytes* vide.

`base64.decodebytes(s)`

Décode un *objet octet-compatible* *s* devant contenir une ou plusieurs lignes de données encodées en base64 et renvoie les `bytes` décodés.

Ajouté dans la version 3.1.

`base64.encode(input, output)`

Encode le contenu du fichier binaire *input* et écrit les données encodées en base64 résultantes dans le fichier *output*. *input et *output* doivent être des *objets fichiers*. *input* est lu jusqu'à ce que `input.readline()` renvoie un objet *bytes* vide. `encode()` insère un caractère de saut de ligne (`b'\n'`) tous les 76 octets de sortie et assure que celle-ci se termine par une nouvelle ligne, comme spécifié par la **RFC 2045** (MIME).

`base64.encodebytes(s)`

Encode un *objet octet-compatible* *s* pouvant contenir des données binaires arbitraires et renvoie les `bytes` contenant les données encodées en base64. Un caractère de saut de ligne (`b'\n'`) est inséré tous les 76 octets de sortie et celle-ci se termine par une nouvelle ligne, comme spécifié par la **RFC 2045** (MIME).

Ajouté dans la version 3.1.

Un exemple d'utilisation du module :

```
>>> import base64
>>> encoded = base64.b64encode(b'data to be encoded')
>>> encoded
b'ZGF0YSB0byBiZSBlbmNvZGVk'
>>> data = base64.b64decode(encoded)
>>> data
b'data to be encoded'
```

20.5.1 Considérations de sécurité

A new security considerations section was added to **RFC 4648** (section 12) ; it's recommended to review the security section for any code deployed to production.

Voir aussi

Module `binascii`
Module secondaire contenant les conversions ASCII vers binaire et binaire vers ASCII.

RFC 1521 — MIME (*Multipurpose Internet Mail Extensions*) *Part One : Mechanisms for Specifying and Describing the Format of Internet Message Bodies*
La Section 5.2, "*Base64 Content-Transfer-Encoding*", donne la définition de l'encodage base64.

20.6 binascii --- Convert between binary and ASCII

The binascii module contains a number of methods to convert between binary and various ASCII-encoded binary representations. Normally, you will not use these functions directly but use wrapper modules like base64 instead. The binascii module contains low-level functions written in C for greater speed that are used by the higher-level modules.

> **Note**
>
> La fonction a2b_* accepte des chaînes de caractères contenant seulement des caractères ASCII. D'autres fonctions acceptent seulement des objets *bytes et similaire* (tel que bytes, bytearray et autres objets qui supportent le protocole tampon).
>
> Modifié dans la version 3.3 : Les chaines de caractères *unicode* seulement composées de caractères ASCII sont désormais acceptées par les fonctions a2b_*.

Le module binascii définit les fonctions suivantes :

binascii.**a2b_uu**(*string*)

Convertit une seule ligne de donnée *uuencoded* en binaire et renvoie la donnée binaire. Les lignes contiennent normalement 45 octets (binaire), sauf pour la dernière ligne. Il se peut que la ligne de donnée soit suivie d'un espace blanc.

binascii.**b2a_uu**(*data*, *, *backtick=False*)

Convertit les données binaires en une ligne de caractères ASCII, la valeur renvoyée est la ligne convertie incluant un caractère de nouvelle ligne. La longueur de *data* doit être au maximum de 45. Si *backtick* est vraie, les zéros sont représentés par '`' plutôt que par des espaces.

Modifié dans la version 3.7 : Ajout du paramètre *backtick*.

binascii.**a2b_base64**(*string*, /, *, *strict_mode=False*)

Convertit un bloc de donnée en *base64* en binaire et renvoie la donnée binaire. Plus d'une ligne peut être passé à la fois.

If *strict_mode* is true, only valid base64 data will be converted. Invalid base64 data will raise binascii.Error.

Valid base64 :
- Conforms to **RFC 3548**.
- Contains only characters from the base64 alphabet.
- Contains no excess data after padding (including excess padding, newlines, etc.).
- Does not start with a padding.

Modifié dans la version 3.11 : Added the *strict_mode* parameter.

binascii.**b2a_base64**(*data*, *, *newline=True*)

Convertit les données binaires en une ligne de caractères ASCII en codage base 64. La valeur de renvoyée et la ligne convertie, incluant un caractère de nouvelle ligne si *newline* est vraie. La sortie de cette fonction se conforme à **RFC 3548**.

Modifié dans la version 3.6 : Ajout du paramètre *newline*.

binascii.**a2b_qp**(*data*, *header=False*)

Convertit un bloc de données *quoted-printable* en binaire et renvoie les données binaires. Plus d'une ligne peut être passée à la fois. Si l'argument optionnel *header* est présent et vrai, les traits soulignés seront décodés en espaces.

binascii.**b2a_qp**(*data*, *quotetabs=False*, *istext=True*, *header=False*)

Convertit les données binaires en ligne(s) de caractères ASCII en codage imprimable entre guillemets. La valeur de retour est lales lignes(s) convertie(s). Si l'argument optionnel *quotetabs* est présent et vrai, toutes les

tabulations et espaces seront encodés. Si l'argument optionnel *istext* est présent et faux, les nouvelles lignes ne sont pas encodées mais les espaces de fin de ligne le seront. Si l'argument optionnel *header* est présent et vrai, les espaces vont être encodés comme de traits soulignés selon **RFC 1522**. Si l'argument optionnel *header* est présent et faux, les caractères de nouvelle ligne seront également encodés ; sinon la conversion de saut de ligne pourrait corrompre le flux de données binaire.

binascii.**crc_hqx**(*data*, *value*)

Calcule une valeur en CRC 16-bit de *data*, commençant par *value* comme CRC initial et renvoie le résultat. Ceci utilise le CRC-CCITT polynomial $x^{16} + x^{12} + x^5 + 1$, souvent représenté comme *0x1021*. Ce CRC est utilisé dans le format *binhex4*.

binascii.**crc32**(*data*[, *value*])

Compute CRC-32, the unsigned 32-bit checksum of *data*, starting with an initial CRC of *value*. The default initial CRC is zero. The algorithm is consistent with the ZIP file checksum. Since the algorithm is designed for use as a checksum algorithm, it is not suitable for use as a general hash algorithm. Use as follows :

```
print(binascii.crc32(b"hello world"))
# Or, in two pieces:
crc = binascii.crc32(b"hello")
crc = binascii.crc32(b" world", crc)
print('crc32 = {:#010x}'.format(crc))
```

Modifié dans la version 3.0 : Renvoie une valeur non-signée.

binascii.**b2a_hex**(*data*[, *sep*[, *bytes_per_sep=1*]])

binascii.**hexlify**(*data*[, *sep*[, *bytes_per_sep=1*]])

Renvoie la représentation hexadécimale du binaire *data*. Chaque octet de *data* est converti en la représentation 2 chiffres correspondante. L'objet octets renvoyé est donc deux fois plus long que la longueur de *data*.

Fonctionnalité similaire est également commodément accessible en utilisant la méthode `bytes.hex()`.

Si *sep* est spécifié, il doit s'agir d'une chaîne de caractères ou d'un objet *byte*. Il sera inséré dans la sortie tous les *bytes_per_sep* octets. Par défaut, l'emplacement du séparateur est compté à partir de l'extrémité droite de la sortie. Si vous souhaitez compter à partir de la gauche, indiquez une valeur *bytes_per_sep* négative.

```
>>> import binascii
>>> binascii.b2a_hex(b'\xb9\x01\xef')
b'b901ef'
>>> binascii.hexlify(b'\xb9\x01\xef', '-')
b'b9-01-ef'
>>> binascii.b2a_hex(b'\xb9\x01\xef', b'_', 2)
b'b9_01ef'
>>> binascii.b2a_hex(b'\xb9\x01\xef', b' ', -2)
b'b901 ef'
```

Modifié dans la version 3.8 : ajout des paramètres *sep* et *bytes_per_sep*.

binascii.**a2b_hex**(*hexstr*)

binascii.**unhexlify**(*hexstr*)

Renvoie la donnée binaire représentée par la chaîne de caractères hexadécimale *hexstr*. Cette fonction est l'inverse de `b2a_hex()`. *hexstr* doit contenir un nombre pair de chiffres hexadécimaux (qui peuvent être en majuscule ou minuscule), sinon une exception `Error` est levée.

Une fonctionnalité similaire (n'acceptant que les arguments de chaîne de texte, mais plus libérale vis-à-vis des espaces blancs) est également accessible en utilisant la méthode de classe `bytes.fromhex()`.

exception binascii.**Error**

Exception levée en cas d'erreurs. Ce sont typiquement des erreurs de programmation.

exception binascii.**Incomplete**

Exception levée par des données incomplète. Il ne s'agit généralement pas d'erreurs de programmation, mais elles peuvent être traitées en lisant un peu plus de données et en réessayant.

Voir aussi

Module `base64`
Support de l'encodage *base64-style* conforme RFC en base 16, 32, 64 et 85.

Module `quopri`
Support de l'encodage *quote-printable* utilisé par les messages *email* MIME.

20.7 `quopri` --- Encode and decode MIME quoted-printable data

Code source : Lib/quopri.py

Ce module effectue des encodages et décodages de transport *quoted-printable*, tel que définis dans la **RFC 1521** : "*MIME (Multipurpose Internet Mail Extensions) Part One Mechanisms for Specifying and Describing the Format of Internet Message Bodies*". L'encodage *quoted-printable* est adapté aux données dans lesquelles peu de données ne sont pas affichables. L'encodage *base64* disponible dans le module `base64` est plus compact dans les cas où ces caractères sont nombreux, typiquement pour encoder des images.

`quopri.decode`(*input*, *output*, *header=False*)

Décode le contenu du fichier *input* et écrit le résultat décodé, binaire, dans le fichier *output*. *input* et *output* doivent être des *objets fichiers binaires*. Si l'argument facultatif *header* est fourni et vrai, les *underscore* seront décodés en espaces. C'est utilisé pour décoder des entêtes encodées "Q" décrits dans la RFC **RFC 1522** : "*MIME (Multipurpose Internet Mail Extensions) Part Two : Message Header Extensions for Non-ASCII Text*".

`quopri.encode`(*input*, *output*, *quotetabs*, *header=False*)

Encode le contenu du fichier *input* et écrit le résultat dans le fichier *output*. *input* et *output* doivent être des *objets fichiers binaires*. *quotetabs* (paramètre obligatoire) permet de choisir le style d'encodage des espaces et des tabulations, si vrai les espaces seront encodées, sinon elles seront laissées inchangées. Notez que les espaces et tabulations en fin de ligne sont toujours encodées, tel que spécifié par la **RFC 1521**. *header* est une option permettant d'encoder les espace en *underscores*, tel que spécifié par la **RFC 1522**.

`quopri.decodestring`(*s*, *header=False*)

Fonctionne comme `decode()`, sauf qu'elle accepte des `bytes` comme source, et renvoie les `bytes` décodés correspondants.

`quopri.encodestring`(*s*, *quotetabs=False*, *header=False*)

Fonctionne comme `encode()`, sauf qu'elle accepte des `bytes` comme source et renvoie les `bytes` encodés correspondants. Par défaut, `False` est donné au paramètre *quotetabs* de la fonction `encode()`.

Voir aussi

Module `base64`
Encode et décode des données MIME en *base64*

CHAPITRE 21

Outils de traitement de balises structurées

Python intègre une variété de modules pour fonctionner avec différentes formes de données structurées et balisées, comme le SGML (*Standard Generalized Markup Language*), le HTML (*Hypert Markup Language*), et quelques interfaces pour travailler avec du XML (*eXtensible Markup Language*).

21.1 `html` --- HyperText Markup Language support

Source code : Lib/html/__init__.py

Ce module définit des outils permettant la manipulation d'HTML.

`html.escape`(*s*, *quote=True*)

Convertit les caractères `&`, `<` et `>` de la chaîne de caractères *s* en séquences HTML valides. À utiliser si le texte à afficher pourrait contenir de tels caractères dans le HTML. Si le paramètre optionnel *quote* est vrai, les caractères (`"`) et (`'`) sont également traduits ; cela est utile pour les inclusions dans des valeurs d'attributs HTML délimitées par des guillemets, comme dans `<a href="...">`.

Ajouté dans la version 3.2.

`html.unescape`(*s*)

Convertit toutes les références de caractères nommés et numériques (e.g. `>`, `>`, `>`) dans la chaîne de caractères *s* par les caractères Unicode correspondants. Cette fonction utilise les règles définies par le standard HTML 5 à la fois pour les caractères valides et les caractères invalides, et la *liste des références des caractères nommés en HTML 5*.

Ajouté dans la version 3.4.

Les sous-modules dans le paquet `html` sont :

- *`html.parser`* -- Parseur HTML/XHTML avec un mode de *parsing* tolérant
- *`html.entities`* -- Définitions d'entités HTML

21.2 `html.parser` --- Simple HTML and XHTML parser

Code source : Lib/html/parser.py

Ce module définit une classe *HTMLParser* qui sert de base pour l'analyse syntaxique de fichiers texte formatés HTML (*HyperText Mark-up Language*, le « langage de balisage hypertexte ») et XHTML (*EXtensible HyperText Markup Language*, le « langage extensible de balisage hypertexte »).

class html.parser.**HTMLParser**(*, *convert_charrefs=True*)

Crée une instance d'analyseur capable de traiter tout balisage, même invalide.

Si *convert_charrefs* est `True` (valeur par défaut), toute référence de caractère (sauf ceux enchâssés dans des éléments `script`/`style`) est automatiquement convertie en son caractère Unicode.

Une instance de *HTMLParser* est alimentée par des données HTML. Elle fait appel à des méthodes offrant un traitement spécifique quand est rencontré un élément de balisage : balise ouvrante ou fermante, textes, commentaires… Pour implémenter le comportement désiré, l'utilisateur crée une sous-classe de *HTMLParser* en surchargeant ses méthodes.

Cet analyseur ne vérifie ni que les balises fermantes correspondent aux balises ouvrantes, ni n'invoque le gestionnaire de balises fermantes pour les éléments implicitement fermés par un élément extérieur.

Modifié dans la version 3.4 : L'argument *convert_charrefs* a été ajouté.

Modifié dans la version 3.5 : La valeur par défaut de l'argument *convert_charrefs* est désormais `True`.

21.2.1 Exemple d'application de l'analyseur HTML

Comme exemple simple, un analyseur HTML minimal qui utilise la classe *HTMLParser* pour afficher les balises ouvrantes, les balises fermantes ainsi que les données quand elles apparaissent :

```
from html.parser import HTMLParser

class MyHTMLParser(HTMLParser):
    def handle_starttag(self, tag, attrs):
        print("Encountered a start tag:", tag)

    def handle_endtag(self, tag):
        print("Encountered an end tag :", tag)

    def handle_data(self, data):
        print("Encountered some data  :", data)

parser = MyHTMLParser()
parser.feed('<html><head><title>Test</title></head>'
            '<body><h1>Parse me!</h1></body></html>')
```

La sortie est alors :

```
Encountered a start tag: html
Encountered a start tag: head
Encountered a start tag: title
Encountered some data  : Test
Encountered an end tag : title
Encountered an end tag : head
Encountered a start tag: body
Encountered a start tag: h1
Encountered some data  : Parse me!
Encountered an end tag : h1
Encountered an end tag : body
Encountered an end tag : html
```

21.2.2 Méthodes de la classe `HTMLParser`

Les instances de `HTMLParser` disposent des méthodes suivantes :

`HTMLParser.feed(data)`

Alimente l'analyseur avec du texte. Ce texte est traité dans la mesure où il constitue des éléments complets ; les données incomplètes sont mises dans un tampon jusqu'à ce que d'autres données soient fournies ou que la méthode `close()` soit appelée. L'argument *data* doit être de classe `str`.

`HTMLParser.close()`

Force le traitement de toutes les données du tampon comme si elles étaient suivies par un caractère *fin de fichier*. Cette méthode peut-être redéfinie par une classe dérivée pour ajouter des traitements supplémentaires à la fin de l'entrée, mais la version redéfinie devra impérativement appeler la méthode `close()` de la classe de base `HTMLParser`.

`HTMLParser.reset()`

Réinitialise l'instance. Toutes les données non traitées sont perdues. Cette méthode est appelée implicitement lors de l'instanciation.

`HTMLParser.getpos()`

Renvoie le numéro de ligne et le numéro du caractère dans la ligne où le curseur est positionné.

`HTMLParser.get_starttag_text()`

Return the text of the most recently opened start tag. This should not normally be needed for structured processing, but may be useful in dealing with HTML "as deployed" or for re-generating input with minimal changes (whitespace between attributes can be preserved, etc.).

Les méthodes suivantes sont appelées lors de la rencontre de données ou d'éléments de balisage ; elles sont destinées à être surchargées par la sous-classe. L'implémentation de la classe de base ne fait rien (sauf pour ce qui est de `handle_startendtag()`) :

`HTMLParser.handle_starttag(tag, attrs)`

Cette méthode est appelée pour traiter une balise ouvrante (p. ex. `<div id="main">`).

L'argument *tag* contient le nom de la balise en minuscules. L'argument *attrs* contient une liste de *n-uplets* `(name, value)` regroupant les attributs présents entre les symboles `<` et `>` de la balise. Le paramètre *name* est converti en minuscule ; les guillemets sont supprimés du paramètre *value* et toute entité de référence ou de caractère est remplacée.

Par exemple, pour la balise `<A HREF="https://www.cwi.nl/">`, cette méthode est appelée par `handle_starttag('a', [('href', 'https://www.cwi.nl/')])`.

Toute référence d'entité présente dans `html.entities` est remplacée dans la valeur des attributs.

`HTMLParser.handle_endtag(tag)`

Cette méthode est appelée pour traiter les balises fermantes (p. ex. `</div>`).

L'argument *tag* est le nom de la balise en minuscules.

`HTMLParser.handle_startendtag(tag, attrs)`

Traitée de façon similaire à `handle_starttag()`, mais appelée quand l'analyseur rencontre une balise vide de type *XHTML* (p. ex. `<img ... />`). Cette méthode peut-être surchargée par les sous-classes demandant cette information lexicale ; l'implémentation par défaut appelle simplement `handle_starttag()` et `handle_endtag()`.

`HTMLParser.handle_data(data)`

Cette méthode est appelée pour traiter toute donnée arbitraire (p. ex. les nœuds textuels ou les contenus de `<script>...</script>` et `<style>...</style>`).

`HTMLParser.handle_entityref(name)`

Cette méthode est appelée pour traiter les références nommées de caractères de la forme `&name;` (p. ex. `>`), où *name* est une référence à une entité générique (p. ex. `'gt'`). Cette méthode n'est jamais appelée si *convert_charrefs* vaut `True`.

`HTMLParser.handle_charref(name)`

This method is called to process decimal and hexadecimal numeric character references of the form `&#NNN;` and `&#xNNN;`. For example, the decimal equivalent for `>` is `>`, whereas the hexadecimal is `>` ; in this case the method will receive `'62'` or `'x3E'`. This method is never called if *convert_charrefs* is `True`.

`HTMLParser.handle_comment(data)`

Cette méthode est appelée quand un commentaire (p. ex. `<!--commentaire-->`) est rencontré.

Par exemple, le commentaire `<!-- commentaire -->` provoque l'appel de cette méthode avec l'argument `' commentaire '`.

Le contenu des commentaires conditionnels d'Internet Explorer (*condcoms*) ne sont pas traités de manière particulière par l'analyseur. Ils sont passés à cette méthode comme tous les autres commentaires. Ainsi, pour `<!--[if IE 9]>Contenu spécifique à IE9<![endif]-->`, cette méthode sera appelée avec `'[if IE 9]>Contenu spécifique à IE9<![endif]'`.

`HTMLParser.handle_decl(decl)`

Cette méthode est appelée pour traiter la déclaration *doctype* de HTML (p. ex. `<!DOCTYPE html>`).

Le paramètre *decl* contient la totalité de la déclaration contenue dans le balisage `<!...>` (p. ex. `'DOCTYPE html'`).

`HTMLParser.handle_pi(data)`

Méthode appelée quand une instruction de traitement est rencontrée. Le paramètre *data* contient la totalité de l'instruction de traitement. Par exemple, pour l'instruction de traitement `<?proc color='rouge'>`, cette méthode est appelée par `handle_pi("proc color='rouge'")`. Elle est destinée à être surchargée dans une classe dérivée ; la classe de base ne réalise aucun traitement.

> **Note**
>
> La classe `HTMLParser` utilise les règles syntaxiques de **SGML** pour traiter les instructions de traitement. Une instruction de traitement *XHTML* utilisant une terminaison en `'?'` se traduit par l'inclusion de ce `'?'` dans *data*.

`HTMLParser.unknown_decl(data)`

Cette méthode est appelée quand une déclaration non reconnue est lue par l'analyseur.

Le paramètre *data* contient toute la déclaration enchâssée dans le balisage `<![...]>`. Il est parfois utile de le surcharger dans une classe dérivée. L'implémentation de la classe de base ne réalise aucune opération.

21.2.3 Exemples

La classe suivante implémente un analyseur qui est utilisé dans les exemples ci-dessous :

```
from html.parser import HTMLParser
from html.entities import name2codepoint

class MyHTMLParser(HTMLParser):
    def handle_starttag(self, tag, attrs):
        print("Start tag:", tag)
        for attr in attrs:
            print("     attr:", attr)

    def handle_endtag(self, tag):
        print("End tag  :", tag)

    def handle_data(self, data):
        print("Data     :", data)
```

(suite sur la page suivante)

(suite de la page précédente)

```
    def handle_comment(self, data):
        print("Comment  :", data)

    def handle_entityref(self, name):
        c = chr(name2codepoint[name])
        print("Named ent:", c)

    def handle_charref(self, name):
        if name.startswith('x'):
            c = chr(int(name[1:], 16))
        else:
            c = chr(int(name))
        print("Num ent  :", c)

    def handle_decl(self, data):
        print("Decl     :", data)

parser = MyHTMLParser()
```

Traitement du *doctype* :

```
>>> parser.feed('<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" '
...             '"http://www.w3.org/TR/html4/strict.dtd">')
Decl     : DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/
↪html4/strict.dtd"
```

Analyse d'un élément avec un titre et des attributs :

```
>>> parser.feed('<img src="python-logo.png" alt="The Python logo">')
Start tag: img
     attr: ('src', 'python-logo.png')
     attr: ('alt', 'The Python logo')
>>>
>>> parser.feed('<h1>Python</h1>')
Start tag: h1
Data     : Python
End tag  : h1
```

Le contenu des éléments `script` et `style` est renvoyé tel quel (sans autre traitement) :

```
>>> parser.feed('<style type="text/css">#python { color: green }</style>')
Start tag: style
     attr: ('type', 'text/css')
Data     : #python { color: green }
End tag  : style

>>> parser.feed('<script type="text/javascript">'
...             'alert("<strong>hello!</strong>");</script>')
Start tag: script
     attr: ('type', 'text/javascript')
Data     : alert("<strong>hello!</strong>");
End tag  : script
```

Traitement des commentaires :

```
>>> parser.feed('<!-- a comment -->'
...             '<!--[if IE 9]>IE-specific content<![endif]-->')
```

(suite sur la page suivante)

(suite de la page précédente)

```
Comment  :  a comment
Comment  : [if IE 9]>IE-specific content<![endif]
```

L'analyse des caractères nommés et des références numériques de caractères et leur conversion en leur caractère correct (note : ces trois références sont équivalentes à `'>'`) :

```
>>> parser.feed('&gt;&#62;&#x3E;')
Named ent: >
Num ent  : >
Num ent  : >
```

Il est possible de fournir des portions de code incomplètes à *feed()*, mais alors il est possible que *handle_data()* soit appelée plusieurs fois, à moins que *convert_charrefs* soit `True` :

```
>>> for chunk in ['<sp', 'an>buff', 'ered ', 'text</s', 'pan>']:
...     parser.feed(chunk)
...
Start tag: span
Data     : buff
Data     : ered
Data     : text
End tag  : span
```

L'analyse de code *HTML* non valide (p. ex. des attributs sans guillemets) fonctionne également :

```
>>> parser.feed('<p><a class=link href=#main>tag soup</p ></a>')
Start tag: p
Start tag: a
     attr: ('class', 'link')
     attr: ('href', '#main')
Data     : tag soup
End tag  : p
End tag  : a
```

21.3 `html.entities` --- Definitions of HTML general entities

Source code : Lib/html/entities.py

Ce module définit quatre dictionnaires, *html5*, *name2codepoint*, *codepoint2name*, et *entitydefs*.

`html.entities.`**`html5`**

Un dictionnaire qui fait correspondre les références de caractères nommés HTML5 [1] aux caractères Unicode équivalents, e.g. `html5['gt;'] == '>'`. À noter que le point-virgule en fin de chaîne est inclus dans le nom (e.g. `'gt;'`), toutefois certains noms sont acceptés par le standard même sans le point-virgule : dans ce cas, le nom est présent à la fois avec et sans le `;`. Voir aussi *html.unescape()*.

Ajouté dans la version 3.3.

`html.entities.`**`entitydefs`**

Un dictionnaire qui fait correspondre les définitions d'entités XHTML 1.0 avec leur remplacement en ISO Latin-1.

`html.entities.`**`name2codepoint`**

A dictionary that maps HTML4 entity names to the Unicode code points.

1. See https://html.spec.whatwg.org/multipage/named-characters.html#named-character-references

html.entities.**codepoint2name**

A dictionary that maps Unicode code points to HTML4 entity names.

Notes

21.4 Modules de traitement XML

Code source : Lib/xml/

Les interfaces de Python de traitement de XML sont regroupées dans le paquet `xml`.

> **Avertissement**
>
> les modules XML ne sont pas protégés contre les données mal construites ou malveillantes. Si vous devez parcourir des données douteuses non authentifiées, lisez les sections *Vulnérabilités XML* et *The defusedxml Package*.

Il est important de noter que les modules dans le paquet `xml` nécessitent qu'au moins un analyseur compatible *SAX* soit disponible. L'analyseur *Expat* est inclus dans Python, ainsi le module `xml.parsers.expat` est toujours disponible.

La documentation des interfaces vers *DOM* et *SAX* se trouve dans `xml.dom` et `xml.sax`.

Les sous-modules de traitement XML sont :

- `xml.etree.ElementTree` : l'API *ElementTree*, un processeur simple et léger
- `xml.dom` : la définition de l'API DOM
- `xml.dom.minidom` : une implémentation minimale de DOM
- `xml.dom.pulldom` : gestion de la construction partielle des arbres DOM
- `xml.sax` : classes mères *SAX2* et fonctions utilitaires
- `xml.parsers.expat` : l'interface de l'analyseur *Expat*

21.4.1 Vulnérabilités XML

Les modules de traitement XML ne sont pas sécurisés contre les données construites malicieusement. Un attaquant peut abuser des fonctionnalités XML pour exécuter des attaques par déni de service, accéder à des fichiers locaux, générer des connexions réseaux à d'autres machines ou contourner des pare-feux.

Le tableau suivant donne une vue d'ensemble des attaques connues et indique si les différents modules y sont vulnérables.

type	*sax*	*etree*	*minidom*	*pulldom*	*xmlrpc*
billion laughs	**Vulnérable** (1)	**Vulnérable** (1)	**Vulnérable** (1)	**Vulnérable** (1)	**Vulnérable** (1)
quadratic blowup	**Vulnérable** (1)	**Vulnérable** (1)	**Vulnérable** (1)	**Vulnérable** (1)	**Vulnérable** (1)
external entity expansion	Sûr (5)	Sûr (2)	Sûr (3)	Sûr (5)	Sûr (4)
Récupération de DTD	Sûr (5)	Sûr	Sûr	Sûr (5)	Sûr
decompression bomb	Sûr	Sûr	Sûr	Sûr	**Vulnérable**
large tokens	**Vulnerable** (6)	**Vulnerable** (6)	**Vulnerable** (6)	**Vulnerable** (6)	**Vulnerable** (6)

1. Expat 2.4.1 and newer is not vulnerable to the "billion laughs" and "quadratic blowup" vulnerabilities. Items still listed as vulnerable due to potential reliance on system-provided libraries. Check `pyexpat.EXPAT_VERSION`.
2. `xml.etree.ElementTree` doesn't expand external entities and raises a `ParseError` when an entity occurs.

3. `xml.dom.minidom` n'étend pas les entités externes et renvoie simplement l'entité mot pour mot (non étendue).
4. `xmlrpc.client` doesn't expand external entities and omits them.
5. Depuis Python 3.7.1, par défaut les entités générales externes ne sont plus traitées.
6. Expat 2.6.0 and newer is not vulnerable to denial of service through quadratic runtime caused by parsing large tokens. Items still listed as vulnerable due to potential reliance on system-provided libraries. Check `pyexpat.EXPAT_VERSION`.

billion laughs / exponential entity expansion
: L'attaque Billion Laughs – aussi connue comme *exponential entity expansion* – utilise de multiples niveaux d'entités imbriquées. Chaque entité se réfère à une autre entité de multiple fois. L'entité finale contient une chaîne courte. Le résultat de l'expansion exponentielle génère plusieurs gigaoctets de texte et consomme beaucoup de mémoire et de temps processeur.

quadratic blowup entity expansion
: Une attaque *quadratic blowup* est similaire à l'attaque Billion Laughs ; il s'agit également d'un abus d'extension d'entités. Au lieu d'utiliser des entités imbriquées, cette attaque répète encore et encore une seule entité de plusieurs milliers de caractères. Cette attaque n'est pas aussi efficace que la version exponentielle mais contourne les contre-mesures de l'analyseur qui interdit les entités imbriquées de multiples fois.

external entity expansion
: Les déclarations d'entités peuvent contenir plus que du texte de substitution. Elles peuvent également référencer des ressources externes ou des fichiers locaux. L'analyseur XML accède à ces fichiers et inclut les contenus dans le document XML.

Récupération de DTD
: Certaines bibliothèques XML comme `xml.dom.pulldom` de Python récupère les documents de définitions de types (*DTD*) depuis des emplacements distants ou locaux. La fonctionnalité a des implications similaires au problème d'extension d'entités externes.

decompression bomb
: Des bombes de décompression (ou ZIP bomb) sont valables pour toutes les bibliothèques XML qui peuvent analyser des flux XML compressés comme des flux HTTP *gzip* ou des fichiers compressés *LZMA*. Pour L'attaquant, cela permet de réduire d'une magnitude d'ordre 3 ou plus la quantité de données transmises.

large tokens
: Expat needs to re-parse unfinished tokens ; without the protection introduced in Expat 2.6.0, this can lead to quadratic runtime that can be used to cause denial of service in the application parsing XML. The issue is known as **CVE 2023-52425**.

The documentation for defusedxml on PyPI has further information about all known attack vectors with examples and references.

21.4.2 The `defusedxml` Package

defusedxml is a pure Python package with modified subclasses of all stdlib XML parsers that prevent any potentially malicious operation. Use of this package is recommended for any server code that parses untrusted XML data. The package also ships with example exploits and extended documentation on more XML exploits such as XPath injection.

21.5 `xml.etree.ElementTree` --- The ElementTree XML API

Code Source : Lib/xml/etree/ElementTree.py

Le module `xml.etree.ElementTree` implémente une API simple et efficace pour analyser et créer des données XML.

Modifié dans la version 3.3 : ce module utilise une implémentation rapide chaque fois que c'est possible.

Obsolète depuis la version 3.3 : The `xml.etree.cElementTree` module is deprecated.

> **Avertissement**
>
> le module *xml.sax* n'est pas sécurisé contre les données construites de façon malveillante. Si vous avez besoin d'analyser des données non sécurisées ou non authentifiées, référez-vous à *Vulnérabilités XML*.

21.5.1 Tutoriel

Ceci est un petit tutoriel pour utiliser *xml.etree.ElementTree* (ET). Le but est de montrer quelques composants et les concepts basiques du module.

Arborescence et éléments XML

XML est un format de données fondamentalement hiérarchique et la façon la plus naturelle de le représenter est avec un arbre. ET a deux classes pour ce but : *ElementTree* représente l'ensemble du document XML comme un arbre et *Element* représente un simple nœud dans cet arbre. Les interactions (lire et écrire vers et depuis des fichiers) sur le document sont habituellement effectuées au niveau de *ElementTree*. Les interactions sur un seul élément XML et ses sous-éléments sont effectuées au niveau de *Element*.

Analyse XML

We'll be using the fictive `country_data.xml` XML document as the sample data for this section :

```
<?xml version="1.0"?>
<data>
    <country name="Liechtenstein">
        <rank>1</rank>
        <year>2008</year>
        <gdppc>141100</gdppc>
        <neighbor name="Austria" direction="E"/>
        <neighbor name="Switzerland" direction="W"/>
    </country>
    <country name="Singapore">
        <rank>4</rank>
        <year>2011</year>
        <gdppc>59900</gdppc>
        <neighbor name="Malaysia" direction="N"/>
    </country>
    <country name="Panama">
        <rank>68</rank>
        <year>2011</year>
        <gdppc>13600</gdppc>
        <neighbor name="Costa Rica" direction="W"/>
        <neighbor name="Colombia" direction="E"/>
    </country>
</data>
```

Nous pouvons importer ces données en lisant un fichier :

```
import xml.etree.ElementTree as ET
tree = ET.parse('country_data.xml')
root = tree.getroot()
```

Ou depuis une chaîne de caractères :

```
root = ET.fromstring(country_data_as_string)
```

fromstring() analyse les données XML depuis une chaîne de caractères et produit un *Element*, qui est l'élément racine de l'arbre analysé. D'autres fonctions d'analyse peuvent créer un *ElementTree*. Vérifiez la documentation

pour en être sûr.

En tant qu'`Element`, `root` a une balise (*tag*) et un dictionnaire d'attributs :

```
>>> root.tag
'data'
>>> root.attrib
{}
```

Il contient aussi des nœuds enfants sur lesquels nous pouvons itérer :

```
>>> for child in root:
...     print(child.tag, child.attrib)
...
country {'name': 'Liechtenstein'}
country {'name': 'Singapore'}
country {'name': 'Panama'}
```

Les enfants sont imbriqués et nous pouvons accéder aux nœuds enfants spécifiques *via* un indice :

```
>>> root[0][1].text
'2008'
```

Note

les éléments du XML d'entrée ne sont pas tous considérés comme des éléments de l'arborescence. Actuellement, le module ignore les commentaires XML, les instructions de traitements et la déclaration du type de document dans l'entrée. Néanmoins, les arborescences construites en utilisant l'API du module plutôt que par l'analyse du texte XML peuvent contenir des commentaires et des instructions de traitement ; ils seront inclus lors de la génération du XML de sortie. La DTD du document est accessible en passant une instance de `TreeBuilder` au constructeur de `XMLParser`.

API à flux tiré

La plupart des fonctions d'analyse fournies par ce module nécessitent que le document entier soit lu en même temps avant de renvoyer un résultat. Il est possible d'utiliser un `XMLParser` et d'y introduire des données de manière incrémentielle, mais il s'agit d'une API à flux poussé qui appelle des méthodes par rappel automatique, ce qui est de trop bas niveau et peu pratique pour la plupart des besoins. Parfois, ce que l'utilisateur souhaite réellement, c'est pouvoir analyser du XML de manière incrémentielle, sans bloquer les opérations, tout en bénéficiant de la commodité des objets `Element` entièrement construits.

L'outil le plus puissant pour ce faire est `XMLPullParser`. Il ne nécessite pas de lecture bloquante pour obtenir les données XML et est plutôt alimenté en données de manière incrémentielle avec des appels à `XMLPullParser.feed()`. Pour obtenir les éléments XML analysés, appelez `XMLPullParser.read_events()`. Voici un exemple :

```
>>> parser = ET.XMLPullParser(['start', 'end'])
>>> parser.feed('<mytag>sometext')
>>> list(parser.read_events())
[('start', <Element 'mytag' at 0x7fa66db2be58>)]
>>> parser.feed(' more text</mytag>')
>>> for event, elem in parser.read_events():
...     print(event)
...     print(elem.tag, 'text=', elem.text)
...
end
mytag text= sometext more text
```

Le cas d'utilisation évident concerne les applications qui fonctionnent de manière non bloquante dans lesquelles les données XML sont reçues à partir d'un connecteur réseau ou lues de manière incrémentielle à partir d'un périphérique

de stockage. Dans de tels cas, le blocage des lectures est inacceptable.

Parce qu'il est très flexible, `XMLPullParser` peut être peu pratique à utiliser pour des cas d'utilisation plus simples. Si cela ne vous dérange pas que votre application se bloque pendant la lecture de données XML mais que vous souhaitez tout de même disposer de capacités d'analyse incrémentielle, jetez un œil à `iterparse()`. Cela peut être utile lorsque vous lisez un document XML volumineux et que vous ne souhaitez pas le conserver entièrement en mémoire.

Where *immediate* feedback through events is wanted, calling method `XMLPullParser.flush()` can help reduce delay ; please make sure to study the related security notes.

Atteinte d'éléments d'intérêt

`Element` a quelques méthodes très utiles qui aident à parcourir récursivement tous les sous-arbres (ses enfants, leurs enfants et ainsi de suite). Par exemple, `Element.iter()` :

```
>>> for neighbor in root.iter('neighbor'):
...     print(neighbor.attrib)
...
{'name': 'Austria', 'direction': 'E'}
{'name': 'Switzerland', 'direction': 'W'}
{'name': 'Malaysia', 'direction': 'N'}
{'name': 'Costa Rica', 'direction': 'W'}
{'name': 'Colombia', 'direction': 'E'}
```

`Element.findall()` récupère seulement les éléments avec une balise qui sont les descendants directs de l'élément courant. `Element.find()` récupère le *premier* élément avec une balise particulière et `Element.text` accède au contenu textuel de l'élément. `Element.get()` accède aux attributs de l'élément :

```
>>> for country in root.findall('country'):
...     rank = country.find('rank').text
...     name = country.get('name')
...     print(name, rank)
...
Liechtenstein 1
Singapore 4
Panama 68
```

Une caractérisation plus sophistiquée des éléments à rechercher est possible en utilisant *XPath*.

Modification d'un fichier XML

`ElementTree` fournit un moyen simple de créer des documents XML et de les écrire dans des fichiers. La méthode `ElementTree.write()` sert à cet effet.

Une fois créé, un objet `Element` peut être manipulé en changeant directement ses champs (tels que `Element.text`), en ajoutant et en modifiant des attributs (méthode `Element.set()`), ou en ajoutant de nouveaux enfants (par exemple avec `Element.append()`).

Disons que nous voulons en ajouter un au classement de chaque pays, et ajouter un attribut *updated* (« mis à jour ») à l'élément de classement :

```
>>> for rank in root.iter('rank'):
...     new_rank = int(rank.text) + 1
...     rank.text = str(new_rank)
...     rank.set('updated', 'yes')
...
>>> tree.write('output.xml')
```

Maintenant, notre XML ressemble à ceci :

```
<?xml version="1.0"?>
<data>
    <country name="Liechtenstein">
        <rank updated="yes">2</rank>
        <year>2008</year>
        <gdppc>141100</gdppc>
        <neighbor name="Austria" direction="E"/>
        <neighbor name="Switzerland" direction="W"/>
    </country>
    <country name="Singapore">
        <rank updated="yes">5</rank>
        <year>2011</year>
        <gdppc>59900</gdppc>
        <neighbor name="Malaysia" direction="N"/>
    </country>
    <country name="Panama">
        <rank updated="yes">69</rank>
        <year>2011</year>
        <gdppc>13600</gdppc>
        <neighbor name="Costa Rica" direction="W"/>
        <neighbor name="Colombia" direction="E"/>
    </country>
</data>
```

Nous pouvons supprimer des éléments en utilisant `Element.remove()`. Supposons que nous souhaitions supprimer tous les pays dont le classement est supérieur à 50 :

```
>>> for country in root.findall('country'):
...     # using root.findall() to avoid removal during traversal
...     rank = int(country.find('rank').text)
...     if rank > 50:
...         root.remove(country)
...
>>> tree.write('output.xml')
```

Notez que les modifications simultanées lors de l'itération peuvent entraîner des problèmes, tout comme lors de l'itération et de la modification de listes ou de dictionnaires Python. Par conséquent, l'exemple collecte d'abord tous les éléments correspondants avec `root.findall()`, et après seulement parcourt la liste des correspondances.

Maintenant, notre XML ressemble à ceci :

```
<?xml version="1.0"?>
<data>
    <country name="Liechtenstein">
        <rank updated="yes">2</rank>
        <year>2008</year>
        <gdppc>141100</gdppc>
        <neighbor name="Austria" direction="E"/>
        <neighbor name="Switzerland" direction="W"/>
    </country>
    <country name="Singapore">
        <rank updated="yes">5</rank>
        <year>2011</year>
        <gdppc>59900</gdppc>
        <neighbor name="Malaysia" direction="N"/>
    </country>
</data>
```

Création de documents XML

La fonction `SubElement()` fournit également un moyen pratique de créer de nouveaux sous-éléments pour un élément donné :

```
>>> a = ET.Element('a')
>>> b = ET.SubElement(a, 'b')
>>> c = ET.SubElement(a, 'c')
>>> d = ET.SubElement(c, 'd')
>>> ET.dump(a)
<a><b /><c><d /></c></a>
```

Analyse d'un XML avec des espaces de noms

Si l'entrée XML a des espaces de noms, les balises et les attributs avec des préfixes sous la forme *préfixe :unebalise* sont étendus à `{uri}unebalise` où le *préfixe* est remplacé par l'*URI* complet. De plus, s'il existe un espace de noms par défaut, cet URI complet est ajouté au début de toutes les balises non préfixées.

Voici un exemple XML qui intègre deux espaces de noms, l'un avec le préfixe *fictional* et l'autre servant d'espace de noms par défaut :

```
<?xml version="1.0"?>
<actors xmlns:fictional="http://characters.example.com"
        xmlns="http://people.example.com">
    <actor>
        <name>John Cleese</name>
        <fictional:character>Lancelot</fictional:character>
        <fictional:character>Archie Leach</fictional:character>
    </actor>
    <actor>
        <name>Eric Idle</name>
        <fictional:character>Sir Robin</fictional:character>
        <fictional:character>Gunther</fictional:character>
        <fictional:character>Commander Clement</fictional:character>
    </actor>
</actors>
```

Une façon de rechercher et d'explorer cet exemple XML consiste à ajouter manuellement l'URI à chaque balise ou attribut dans le *xpath* d'une recherche `find()` ou `findall()` :

```
root = fromstring(xml_text)
for actor in root.findall('{http://people.example.com}actor'):
    name = actor.find('{http://people.example.com}name')
    print(name.text)
    for char in actor.findall('{http://characters.example.com}character'):
        print(' |-->', char.text)
```

Une meilleure façon de rechercher dans l'exemple XML avec espace de noms consiste à créer un dictionnaire avec vos propres préfixes et à les utiliser dans les fonctions de recherche :

```
ns = {'real_person': 'http://people.example.com',
      'role': 'http://characters.example.com'}

for actor in root.findall('real_person:actor', ns):
    name = actor.find('real_person:name', ns)
    print(name.text)
    for char in actor.findall('role:character', ns):
        print(' |-->', char.text)
```

Ces deux approches donnent le même résultat :

```
John Cleese
 |--> Lancelot
 |--> Archie Leach
Eric Idle
 |--> Sir Robin
 |--> Gunther
 |--> Commander Clement
```

21.5.2 Prise en charge de *XPath*

Ce module fournit une prise en charge limitée des expressions XPath pour localiser des éléments dans une arborescence. L'objectif est de prendre en charge un petit sous-ensemble de la syntaxe abrégée ; un moteur *XPath* complet sort du cadre du module.

Exemple

Voici un exemple qui démontre certaines des fonctionnalités *XPath* du module. Nous utilisons le document XML `countrydata` de la section *Parsing XML* :

```
import xml.etree.ElementTree as ET

root = ET.fromstring(countrydata)

# Top-level elements
root.findall(".")

# All 'neighbor' grand-children of 'country' children of the top-level
# elements
root.findall("./country/neighbor")

# Nodes with name='Singapore' that have a 'year' child
root.findall(".//year/..[@name='Singapore']")

# 'year' nodes that are children of nodes with name='Singapore'
root.findall(".//*[@name='Singapore']/year")

# All 'neighbor' nodes that are the second child of their parent
root.findall(".//neighbor[2]")
```

Pour du XML avec des espaces de noms, utilisez la notation qualifiée habituelle `{namespace}tag` :

```
# All dublin-core "title" tags in the document
root.findall(".//{http://purl.org/dc/elements/1.1/}title")
```

Prise en charge de la syntaxe *XPath*

Syntaxe	Signification
`tag`	Sélectionne tous les éléments enfants avec une balise donnée. Par exemple, `spam` sélectionne tous les éléments enfants nommés `spam` et `spam/egg` sélectionne tous les petits-enfants nommés `egg` dans les enfants nommés `spam`. `{namespace}*` sélectionne toutes les balises dans l'espace de noms donné, `{*}spam` sélectionne les balises nommées `spam` dans n'importe quel (ou aucun) espace de noms et `{}*` sélectionne seulement les balises qui ne sont pas dans un espace de noms. Modifié dans la version 3.8 : la prise en charge des caractères génériques étoiles a été ajoutée.
`*`	Sélectionne tous les éléments enfants, y compris les commentaires et les instructions de traitement. Par exemple, `*/egg` sélectionne tous les petits-enfants nommés `egg`.
`.`	Sélectionne le nœud actuel. C'est surtout utile au début du chemin, pour indiquer qu'il s'agit d'un chemin relatif.
`//`	Sélectionne tous les sous-éléments, à tous les niveaux situés sous l'élément actuel. Par exemple, `.//egg` sélectionne tous les éléments `egg` dans l'arborescence entière.
`..`	Sélectionne l'élément parent. Renvoie `None` si le chemin tente d'atteindre les ancêtres de l'élément de départ (l'élément sur lequel `find` a été appelé).
`[@attrib]`	Sélectionne tous les éléments qui ont l'attribut donné.
`[@attrib='value']`	Sélectionne tous les éléments pour lesquels l'attribut donné a la valeur donnée. La valeur ne peut pas contenir de guillemet simple (`'`).
`[@attrib!='value']`	Sélectionne tous les éléments pour lesquels l'attribut donné n'a pas la valeur donnée. La valeur ne peut pas contenir de guillemet simple (`'`). Ajouté dans la version 3.10.
`[tag]`	Sélectionne tous les éléments qui ont un enfant nommé `tag`. Seuls les enfants immédiats sont pris en charge.
`[.='text']`	Sélectionne tous les éléments dont le contenu textuel complet, y compris les descendants, est égal au `text` donné. Ajouté dans la version 3.7.
`[.!='text']`	Sélectionne tous les éléments dont le contenu textuel complet, y compris les descendants, n'est pas égal au `text` donné. Ajouté dans la version 3.10.
`[tag='text']`	Sélectionne tous les éléments qui ont un enfant nommé `tag` dont le contenu textuel complet, y compris les descendants, est égal au `text` donné.
`[tag!='text']`	Sélectionne tous les éléments qui ont un enfant nommé `tag` dont le contenu textuel complet, y compris les descendants, n'est pas égal au `text` donné. Ajouté dans la version 3.10.
`[position]`	Sélectionne tous les éléments situés à la position donnée. La position peut être soit un entier (1 est la première position), l'expression `last()` (pour la dernière position), ou une position relative à la dernière position (par exemple `last()-1`).

Les prédicats (expressions entre crochets) doivent être précédés d'un nom de balise, d'un astérisque ou d'un autre prédicat. Les prédicats `position` doivent être précédés d'un nom de balise.

21.5.3 Référence

Fonctions

`xml.etree.ElementTree.`**`canonicalize`**(*xml_data=None*, *, *out=None*, *from_file=None*, ***options*)

Fonction de transformation C14N 2.0.

Canonicalization is a way to normalise XML output in a way that allows byte-by-byte comparisons and digital signatures. It reduces the freedom that XML serializers have and instead generates a more constrained XML representation. The main restrictions regard the placement of namespace declarations, the ordering of attributes, and ignorable whitespace.

Cette fonction prend une chaîne de données XML (*xml_data*) ou un chemin de fichier ou un objet de type

fichier (*from_file*) en entrée, la convertit sous la forme canonique et l'écrit à l'aide de l'objet simili-fichier *out*, s'il est fourni, ou le renvoie sous forme de chaîne de texte sinon. Le fichier de sortie reçoit du texte, pas des octets. Il doit donc être ouvert en mode texte avec l'encodage `utf-8`.

Utilisation typique :

```
xml_data = "<root>...</root>"
print(canonicalize(xml_data))

with open("c14n_output.xml", mode='w', encoding='utf-8') as out_file:
    canonicalize(xml_data, out=out_file)

with open("c14n_output.xml", mode='w', encoding='utf-8') as out_file:
    canonicalize(from_file="inputfile.xml", out=out_file)
```

Les *options* de configuration sont les suivantes :

— *with_comments* : définissez cet attribut à vrai pour inclure les commentaires (par défaut : faux)

— ***strip_text* : définissez à vrai pour supprimer les espaces avant et après le contenu du texte** (par défaut : faux)

— ***rewrite_prefixes* : définissez à vrai pour remplacer les préfixes d'espace de noms par « n{nombre} »** (par défaut : faux)

— ***qname_aware_tags* : ensemble de noms de balises prenant en charge les noms qualifiés dans lesquels les préfixes** doivent être remplacés dans le contenu du texte (par défaut : vide)

— ***qname_aware_attrs* : ensemble de noms d'attributs prenant en charge les noms qualifiés dans lesquels les préfixes** doivent être remplacés dans le contenu du texte (par défaut : vide)

— *exclude_attrs* : ensemble de noms d'attributs qui ne doivent pas être sérialisés

— *exclude_tags* : ensemble de noms de balises qui ne doivent pas être sérialisés

Dans la liste d'options ci-dessus, un « ensemble » fait référence à toute collection ou itérable de chaînes, aucun ordre n'est attendu.

Ajouté dans la version 3.8.

`xml.etree.ElementTree.`**`Comment`**(*text=None*)

Fabrique d'éléments de commentaire. Cette fonction crée un élément spécial qui sera sérialisé sous forme de commentaire XML par le sérialiseur standard. La chaîne de commentaire peut être une chaîne d'octets ou une chaîne Unicode. *text* est une chaîne contenant la chaîne de commentaire. Renvoie une instance d'élément représentant un commentaire.

Notez que `XMLParser` ignore les commentaires dans l'entrée au lieu de créer des objets de commentaire pour eux. Un `ElementTree` ne contiendra que des nœuds de commentaires qui ont été insérés dans l'arborescence à l'aide de l'une des méthodes d'`Element`.

`xml.etree.ElementTree.`**`dump`**(*elem*)

Écrit une arborescence d'éléments ou une structure d'éléments dans *sys.stdout*. Cette fonction doit être utilisée uniquement pour le débogage.

Le format de sortie exact dépend de l'implémentation. Dans cette version, il est écrit sous la forme d'un fichier XML ordinaire.

elem est un élément de l'arborescence ou un élément individuel.

Modifié dans la version 3.8 : la fonction `dump()` préserve désormais l'ordre des attributs spécifié par l'utilisateur.

`xml.etree.ElementTree.`**`fromstring`**(*text*, *parser=None*)

Analyse une section XML à partir d'une constante de chaîne. Identique à `XML()`. *text* est une chaîne contenant des données XML. *parser* est une instance d'analyseur facultative. S'il n'est pas fourni, l'analyseur standard `XMLParser` est utilisé. Renvoie une instance d'`Element`.

`xml.etree.ElementTree.`**`fromstringlist`**(*sequence*, *parser=None*)

Analyse un document XML à partir d'une séquence de fragments de chaîne. *séquence* est une liste ou une autre

séquence contenant des fragments de données XML. *parser* est une instance d'analyseur facultative. S'il n'est pas fourni, l'analyseur standard `XMLParser` est utilisé. Renvoie une instance d'`Element`.

Ajouté dans la version 3.2.

`xml.etree.ElementTree.`**`indent`**(*tree*, *space=' '*, *level=0*)

Ajoute des espaces au sous-arbre pour indenter visuellement l'arborescence. Cela peut être utilisé pour générer une sortie XML avec une mise en forme agréable. *tree* peut être un élément ou un *ElementTree*. *espace* est la chaîne d'espaces qui est insérée pour chaque niveau d'indentation, deux caractères d'espace par défaut. Pour indenter des sous-arbres partiels à l'intérieur d'un arbre déjà indenté, transmettez le niveau d'indentation initial dans *level*.

Ajouté dans la version 3.9.

`xml.etree.ElementTree.`**`iselement`**(*element*)

Vérifie si un objet semble être un objet élément valide. *element* est une instance d'élément. Renvoie `True` s'il s'agit d'un objet élément.

`xml.etree.ElementTree.`**`iterparse`**(*source*, *events=None*, *parser=None*)

Parses an XML section into an element tree incrementally, and reports what's going on to the user. *source* is a filename or *file object* containing XML data. *events* is a sequence of events to report back. The supported events are the strings `"start"`, `"end"`, `"comment"`, `"pi"`, `"start-ns"` and `"end-ns"` (the "ns" events are used to get detailed namespace information). If *events* is omitted, only `"end"` events are reported. *parser* is an optional parser instance. If not given, the standard `XMLParser` parser is used. *parser* must be a subclass of `XMLParser` and can only use the default `TreeBuilder` as a target. Returns an *iterator* providing `(event, elem)` pairs ; it has a `root` attribute that references the root element of the resulting XML tree once *source* is fully read. The iterator has the `close()` method that closes the internal file object if *source* is a filename.

Notez que même si `iterparse()` construit l'arborescence de manière incrémentielle, il bloque pendant les lectures sur *source*. En tant que tel, il ne convient pas aux applications dans lesquelles le blocage pendant les lectures ne peut pas être effectué. Pour une analyse entièrement non bloquante, voir `XMLPullParser`.

> **Note**
>
> `iterparse()` garantit uniquement qu'il a vu le caractère `">"` d'une balise de début lorsqu'il émet un événement `"start"`, donc les attributs sont définis mais le contenu des attributs *text* et *tail* n'est pas défini à ce stade. La même chose s'applique aux nœuds enfants de l'élément ; ils peuvent être présents ou non.
>
> Si vous avez besoin d'un élément entièrement rempli, recherchez plutôt les événements `"end"`.

Obsolète depuis la version 3.4 : l'argument *parser*.

Modifié dans la version 3.8 : les évènements `comment` et `pi` ont été ajoutés.

Modifié dans la version 3.13 : Added the `close()` method.

`xml.etree.ElementTree.`**`parse`**(*source*, *parser=None*)

Analyse une section XML dans une arborescence d'éléments. *source* est un nom de fichier ou un objet fichier contenant des données XML. *parser* (facultatif) est une instance d'analyseur. S'il n'est pas fourni, l'analyseur standard `XMLParser` est utilisé. Renvoie une instance d'`ElementTree`.

`xml.etree.ElementTree.`**`ProcessingInstruction`**(*target*, *text=None*)

Fabrique d'éléments *Processing Instruction* (PI). Cette fonction crée un élément spécial qui sera sérialisé en tant qu'instruction de traitement XML. *target* est une chaîne contenant la cible PI. *text* est une chaîne contenant le contenu de l'instruction de traitement (ou PI), s'il est fourni. Renvoie une instance d'élément, représentant l'instruction de traitement.

Note that `XMLParser` skips over processing instructions in the input instead of creating PI objects for them. An `ElementTree` will only contain processing instruction nodes if they have been inserted into to the tree using one of the `Element` methods.

`xml.etree.ElementTree.register_namespace`(*prefix*, *uri*)

Enregistre un préfixe d'espace de noms. Le registre est global et toute correspondance existante pour le préfixe donné ainsi que l'URI de l'espace de noms sera supprimé. *prefix* est un préfixe d'espace de noms. *uri* est un URI d'espace de noms. Les balises et les attributs de cet espace de noms seront sérialisés avec le préfixe donné, si possible.

Ajouté dans la version 3.2.

`xml.etree.ElementTree.SubElement`(*parent*, *tag*, *attrib={}*, ***extra*)

Fabrique de sous-élément. Cette fonction crée une instance d'élément et l'ajoute à un élément existant.

Le nom de l'élément, les noms d'attributs et les valeurs d'attributs peuvent être des chaînes d'octets ou des chaînes Unicode. *parent* est l'élément parent. *tag* est le nom du sous-élément. *attrib* est un dictionnaire facultatif contenant les attributs de l'élément. *extra* contient des attributs supplémentaires, donnés sous forme d'arguments nommés. Renvoie une instance d'élément.

`xml.etree.ElementTree.tostring`(*element*, *encoding='us-ascii'*, *method='xml'*, ***, *xml_declaration=None*, *default_namespace=None*, *short_empty_elements=True*)

Génère une représentation sous forme de chaîne d'un élément XML, y compris tous les sous-éléments. *element* est une instance de `Element`. *encoding* [1] est l'encodage de sortie (la valeur par défaut est US-ASCII). Utilisez `encoding="unicode"` pour générer une chaîne Unicode (sinon, une chaîne d'octets est générée). *method* vaut `"xml"`, `"html"` ou `"text"` (la valeur par défaut est `"xml"`). *xml_declaration*, *default_namespace* et *short_empty_elements* ont la même signification que dans `ElementTree.write()`. Renvoie une chaîne codée (éventuellement) contenant les données XML.

Modifié dans la version 3.4 : Added the *short_empty_elements* parameter.

Modifié dans la version 3.8 : Added the *xml_declaration* and *default_namespace* parameters.

Modifié dans la version 3.8 : la fonction `tostring()` préserve désormais l'ordre des attributs spécifié par l'utilisateur.

`xml.etree.ElementTree.tostringlist`(*element*, *encoding='us-ascii'*, *method='xml'*, ***, *xml_declaration=None*, *default_namespace=None*, *short_empty_elements=True*)

Génère une représentation sous forme de chaîne d'un élément XML, y compris tous les sous-éléments. *element* est une instance de `Element`. *encoding*[1] est l'encodage de sortie (la valeur par défaut est US-ASCII). Utilisez `encoding="unicode"` pour générer une chaîne Unicode (sinon, une chaîne d'octets est générée). *method* vaut `"xml"`, `"html"` ou `"text"` (la valeur par défaut est `"xml"`). *xml_declaration*, *default_namespace* et *short_empty_elements* ont la même signification que dans `ElementTree.write()`. Renvoie une liste de chaînes codées (éventuellement) contenant les données XML. Elle ne garantit aucune séquence spécifique, seulement `b"".join(tostringlist(element)) == tostring(element)`.

Ajouté dans la version 3.2.

Modifié dans la version 3.4 : Added the *short_empty_elements* parameter.

Modifié dans la version 3.8 : Added the *xml_declaration* and *default_namespace* parameters.

Modifié dans la version 3.8 : la fonction `tostringlist()` préserve désormais l'ordre des attributs spécifié par l'utilisateur.

`xml.etree.ElementTree.XML`(*text*, *parser=None*)

Analyse une section XML à partir d'une constante de chaîne. Cette fonction peut être utilisée pour intégrer des « littéraux XML » dans du code Python. *text* est une chaîne contenant des données XML. *parser*, facultatif, est une instance d'analyseur. S'il n'est pas fourni, l'analyseur standard `XMLParser` est utilisé. Renvoie une instance d'`Element`.

`xml.etree.ElementTree.XMLID`(*text*, *parser=None*)

Analyse une section XML à partir d'une constante de chaîne et renvoie également un dictionnaire qui fait correspondre les identifiants d'éléments aux éléments. *text* est une chaîne contenant des données XML. *parser*,

1. La chaîne de caractères encodée inclue dans la sortie XML doit être conforme aux standards. Par exemple, `"UTF-8"` est valide, mais pas `"UTF8"`. Voir https://www.w3.org/TR/2006/REC-xml11-20060816/#NT-EncodingDecl et https://www.iana.org/assignments/character-sets/character-sets.xhtml.

facultatif, est une instance d'analyseur. S'il n'est pas fourni, l'analyseur standard `XMLParser` est utilisé. Renvoie un *n*-uplet contenant une instance d'`Element` et un dictionnaire.

21.5.4 Prise en charge de *XInclude*

Ce module fournit une prise en charge limitée des directives XInclude, via le module d'assistance `xml.etree.ElementInclude`. Ce module peut être utilisé pour insérer des sous-arbres et des chaînes de texte dans des arborescences d'éléments, en fonction des informations contenues dans l'arborescence.

Exemple

Voici un exemple qui montre l'utilisation du module *XInclude*. Pour inclure un document XML dans le document actuel, utilisez l'élément `{http://www.w3.org/2001/XInclude}include`, définissez l'attribut **parse** sur `"xml"` et utilisez l'attribut **href** pour spécifier le document à inclure.

```
<?xml version="1.0"?>
<document xmlns:xi="http://www.w3.org/2001/XInclude">
  <xi:include href="source.xml" parse="xml" />
</document>
```

Par défaut, l'attribut **href** est traité comme un nom de fichier. Vous pouvez utiliser des chargeurs personnalisés pour remplacer ce comportement. Notez également que l'assistant standard ne prend pas en charge la syntaxe *XPointer*.

Pour traiter ce fichier, chargez-le comme d'habitude et passez l'élément racine au module `xml.etree.ElementTree` :

```
from xml.etree import ElementTree, ElementInclude

tree = ElementTree.parse("document.xml")
root = tree.getroot()

ElementInclude.include(root)
```

Le module *ElementInclude* remplace l'élément `{http://www.w3.org/2001/XInclude}include` par l'élément racine du document **source.xml**. Le résultat pourrait ressembler à ceci :

```
<document xmlns:xi="http://www.w3.org/2001/XInclude">
  <para>This is a paragraph.</para>
</document>
```

Si l'attribut **parse** est omis, sa valeur par défaut est `"xml"`. L'attribut *href* est obligatoire.

Pour inclure un document texte, utilisez l'élément `{http://www.w3.org/2001/XInclude}include` et définissez l'attribut **parse** sur `"text"` :

```
<?xml version="1.0"?>
<document xmlns:xi="http://www.w3.org/2001/XInclude">
  Copyright (c) <xi:include href="year.txt" parse="text" />.
</document>
```

Le résultat pourrait ressembler à ceci :

```
<document xmlns:xi="http://www.w3.org/2001/XInclude">
  Copyright (c) 2003.
</document>
```

21.5.5 Référence

Fonctions

`xml.etree.ElementInclude.default_loader(href, parse, encoding=None)`

Default loader. This default loader reads an included resource from disk. *href* is a URL. *parse* is for parse mode either "xml" or "text". *encoding* is an optional text encoding. If not given, encoding is `utf-8`. Returns the expanded resource. If the parse mode is `"xml"`, this is an `Element` instance. If the parse mode is `"text"`, this is a string. If the loader fails, it can return `None` or raise an exception.

`xml.etree.ElementInclude.include(elem, loader=None, base_url=None, max_depth=6)`

This function expands XInclude directives in-place in tree pointed by *elem*. *elem* is either the root `Element` or an `ElementTree` instance to find such element. *loader* is an optional resource loader. If omitted, it defaults to `default_loader()`. If given, it should be a callable that implements the same interface as `default_loader()`. *base_url* is base URL of the original file, to resolve relative include file references. *max_depth* is the maximum number of recursive inclusions. Limited to reduce the risk of malicious content explosion. Pass `None` to disable the limitation.

Modifié dans la version 3.9 : Added the *base_url* and *max_depth* parameters.

Objets *Element*

class `xml.etree.ElementTree.Element(tag, attrib={}, **extra)`

Classe Element. Cette classe définit l'interface Element et fournit une implémentation de référence de cette interface.

Le nom de l'élément, les noms d'attributs et les valeurs d'attributs peuvent être des chaînes d'octets ou des chaînes Unicode. *tag* est le nom de l'élément. *attrib* est un dictionnaire facultatif contenant les attributs des éléments. *extra* contient des attributs supplémentaires, donnés sous forme d'arguments nommés.

tag

Chaîne identifiant le type de données que cet élément représente (en d'autres termes, le type d'élément).

text

tail

Ces attributs peuvent être utilisés pour contenir des données supplémentaires associées à l'élément. Leurs valeurs sont généralement des chaînes mais peuvent être n'importe quel objet spécifique à une application. Si l'élément est créé à partir d'un fichier XML, l'attribut *text* contient soit le texte entre la balise de début de l'élément et sa première balise enfant ou de fin, soit `None` ; l'attribut *tail* contient soit le texte entre la balise de fin de l'élément et la balise suivante, soit `None`. Pour les données XML

```
<a><b>1<c>2<d/>3</c></b>4</a>
```

les attributs *text* et *tail* de l'élément *a* valent `None`, l'attribut *text* de l'élément *b* vaut `"1"` et *tail* vaut `"4"`, l'attribut *text* de l'élément *c* vaut `"2"` et *tail* vaut `None`, et l'attribut *text* de l'élément *d* vaut `None` et *tail* vaut `"3"`.

Pour récupérer le texte interne d'un élément, voir `itertext()`, par exemple `"".join(element.itertext())`.

Les applications peuvent stocker des objets arbitraires dans ces attributs.

attrib

Dictionnaire contenant les attributs de l'élément. Notez que même si la valeur *attrib* est toujours un véritable dictionnaire Python mutable, une implémentation d'*ElementTree* peut choisir d'utiliser une autre représentation interne et de créer le dictionnaire uniquement si quelqu'un le demande. Pour profiter de telles implémentations, utilisez les méthodes de dictionnaire ci-dessous autant que possible.

Les méthodes dictionnaire-compatibles suivantes traitent les attributs de l'élément.

clear()

Réinitialise un élément. Cette fonction supprime tous les sous-éléments, efface tous les attributs et définit les attributs *text* et *tail* sur `None`.

get(*key*, *default=None*)

Accède à l'attribut de l'élément nommé *key*.

Renvoie la valeur de l'attribut ou *default* si l'attribut n'a pas été trouvé.

items()

Renvoie les attributs de l'élément comme une séquence de paire (nom, valeur). Les attributs sont renvoyés dans un ordre arbitraire.

keys()

Renvoie les noms d'attributs des éléments sous forme de liste. Les noms sont renvoyés dans un ordre arbitraire.

set(*key*, *value*)

Change l'attribut *key* à l'élément *value*.

Les méthodes suivantes traitent les éléments enfants (sous-éléments).

append(*subelement*)

Ajoute l'élément *subelement* à la fin de la liste interne des sous-éléments de cet élément. Lève une `TypeError` si *subelement* n'est pas un `Element`.

extend(*subelements*)

Appends *subelements* from an iterable of elements. Raises `TypeError` if a subelement is not an `Element`.

Ajouté dans la version 3.2.

find(*match*, *namespaces=None*)

Recherche le premier sous-élément correspondant à *match*. *match* peut être un nom de balise ou un *path*. Renvoie une instance d'élément ou `None`. *namespaces* est un tableau de correspondances facultatif de préfixes d'espaces de noms vers des noms complets. Passez `''` comme préfixe pour déplacer tous les noms de balises sans préfixe dans l'expression dans l'espace de noms donné.

findall(*match*, *namespaces=None*)

Recherche tous les sous-éléments correspondants, par nom de balise ou *path*. Renvoie une liste contenant tous les éléments correspondants dans l'ordre du document. *namespaces* est un tableau de correspondances facultatif de préfixes d'espaces de noms vers des noms complets. Passez `''` comme préfixe pour déplacer tous les noms de balises sans préfixe dans l'expression dans l'espace de noms donné.

findtext(*match*, *default=None*, *namespaces=None*)

Recherche le texte du premier sous-élément correspondant à *match*. *match* peut être un nom de balise ou un *path*. Renvoie le contenu textuel du premier élément correspondant, ou *default* si aucun élément n'a été trouvé. Notez que si l'élément correspondant n'a pas de contenu textuel, une chaîne vide est renvoyée. *namespaces* est un tableau de correspondances facultatif de préfixes d'espaces de noms vers des noms complets. Passez `''` comme préfixe pour déplacer tous les noms de balises sans préfixe dans l'expression dans l'espace de noms donné.

insert(*index*, *subelement*)

Insère le *subelement* à la position donnée dans cet élément. Lève une `TypeError` si *subelement* n'est pas un `Element`.

iter(*tag=None*)

Crée un *itérateur* sur l'arbre avec l'élément actuel comme racine. L'itérateur parcourt cet élément et tous les éléments situés en dessous, dans l'ordre du document (profondeur en premier). Si *tag* n'est pas `None` ou `'*'`, seuls les éléments dont la balise est égale à *tag* sont renvoyés par l'itérateur. Si l'arborescence est modifiée au cours de l'itération, le résultat est indéfini.

Ajouté dans la version 3.2.

iterfind(*match*, *namespaces=None*)

Recherche tous les sous-éléments correspondants, par nom de balise ou *path*. Renvoie un itérable donnant tous les éléments correspondants dans l'ordre du document. *namespaces* est un tableau de correspondances facultatif de préfixes d'espaces de noms vers des noms complets.

Ajouté dans la version 3.2.

itertext()

Crée un itérateur de texte. L'itérateur parcourt cet élément et tous les sous-éléments, dans l'ordre du document, et renvoie tout le texte interne.

Ajouté dans la version 3.2.

makeelement(*tag*, *attrib*)

Crée un nouvel objet élément du même type que cet élément. N'appelez pas cette méthode, utilisez plutôt la fonction de fabrique `SubElement()`.

remove(*subelement*)

Supprime le *subelement* de l'élément. Contrairement aux méthodes *find**, cette méthode compare les éléments en fonction de l'identité de l'instance, et non de la valeur ou du contenu de la balise.

Les objets `Element` prennent également en charge les méthodes de type séquence suivantes pour travailler avec des sous éléments : `__delitem__()`, `__getitem__()`, `__setitem__()`, `__len__()`.

Caution : Elements with no subelements will test as `False`. In a future release of Python, all elements will test as `True` regardless of whether subelements exist. Instead, prefer explicit `len(elem)` or `elem is not None` tests. :

```
element = root.find('foo')

if not element:  # careful!
    print("element not found, or element has no subelements")

if element is None:
    print("element not found")
```

Modifié dans la version 3.12 : Testing the truth value of an Element emits `DeprecationWarning`.

Avant Python 3.8, l'ordre de sérialisation des attributs XML des éléments était artificiellement rendu prévisible en classant les attributs par leur nom. Sur la base de l'ordre désormais garanti des dictionnaires, cette réorganisation arbitraire a été supprimée dans Python 3.8 pour préserver l'ordre dans lequel les attributs ont été initialement analysés ou créés par le code utilisateur.

En général, le code utilisateur doit essayer de ne pas dépendre d'un ordre spécifique des attributs, étant donné que la documentation de référence XML exclut explicitement que l'ordre des attributs soit porteur d'information. Le code doit être prêt à gérer tout ordre en entrée. Dans les cas où une sortie XML déterministe est requise, par exemple pour la signature cryptographique ou les ensembles de données de test, la sérialisation canonique est disponible avec la fonction `canonicalize()`.

Dans les cas où la sortie canonique n'est pas applicable mais où un ordre d'attributs spécifique est quand même souhaitable en sortie, le code doit viser à créer les attributs directement dans l'ordre souhaité, afin d'éviter des inadéquations de perception pour les lecteurs du code. Dans les cas où cela est difficile à réaliser, une recette comme ci-dessous peut être appliquée avant la sérialisation pour appliquer un ordre indépendant de la création de l'élément :

```
def reorder_attributes(root):
    for el in root.iter():
        attrib = el.attrib
        if len(attrib) > 1:
            # adjust attribute order, e.g. by sorting
            attribs = sorted(attrib.items())
            attrib.clear()
            attrib.update(attribs)
```

Objets *ElementTree*

class xml.etree.ElementTree.**ElementTree**(*element=None*, *file=None*)

Classe enveloppant *ElementTree*. Cette classe représente une hiérarchie d'éléments entière et ajoute une prise en charge supplémentaire pour la sérialisation vers et depuis le XML standard.

element est l'élément racine. L'arborescence est initialisée avec le contenu du fichier XML *file* s'il est fourni.

_setroot(*element*)

Remplace l'élément racine de cette arborescence. Cela supprime le contenu actuel de l'arborescence et le remplace par l'élément donné. À utiliser avec précaution. *element* est une instance d'élément.

find(*match*, *namespaces=None*)

Comme `Element.find()`, commence à la racine de l'arbre.

findall(*match*, *namespaces=None*)

Identique à `Element.findall()`, en commençant à la racine de l'arborescence.

findtext(*match*, *default=None*, *namespaces=None*)

Identique à `Element.findtext()`, en commençant à la racine de l'arborescence.

getroot()

Renvoie l'élément racine de l'arbre.

iter(*tag=None*)

Crée et renvoie un itérateur d'arborescence pour l'élément racine. L'itérateur parcourt tous les éléments de cette arborescence, dans l'ordre des sections. *tag* est la balise à rechercher (la valeur par défaut est de renvoyer tous les éléments).

iterfind(*match*, *namespaces=None*)

Identique à `Element.iterfind()`, en commençant à la racine de l'arborescence.

Ajouté dans la version 3.2.

parse(*source*, *parser=None*)

Charge une section XML externe dans cette arborescence d'éléments. *source* est un nom de fichier ou un *objet fichier*. *parser*, facultatif, est une instance d'analyseur. S'il n'est pas fourni, l'analyseur standard `XMLParser` est utilisé. Renvoie l'élément racine de la section.

write(*file*, *encoding='us-ascii'*, *xml_declaration=None*, *default_namespace=None*, *method='xml'*, *, *short_empty_elements=True*)

Écrit l'arborescence des éléments dans un fichier, au format XML. *file* est un nom de fichier ou un *objet fichier* ouvert en écriture. *encoding*[page 1364, 1] est l'encodage de sortie (la valeur par défaut est US-ASCII). *xml_declaration* contrôle si une déclaration XML doit être ajoutée au fichier. Utilisez `False` pour jamais, `True` pour toujours, `None` pour seulement si ce n'est pas de l'US-ASCII ou de l'UTF-8 ou de l'Unicode (la valeur par défaut est `None`). *default_namespace* définit l'espace de noms XML par défaut (pour `"xmlns"`). *method* vaut `"xml"`, `"html"` ou `"text"` (la valeur par défaut est `"xml"`). Le paramètre nommé (uniquement) *short_empty_elements* contrôle le formatage des éléments qui ne contiennent aucun contenu. Si c'est `True` (valeur par défaut), ils sont écrits sous la forme d'une seule balise auto-fermée, sinon ils sont écrits sous la forme d'une paire de balises de début-fin.

La sortie est une chaîne (`str`) ou du binaire (`bytes`). C'est contrôlé par l'argument *encoding*. Si *encoding* est `"unicode"`, la sortie est une chaîne ; sinon, c'est binaire. Notez que cela peut entrer en conflit avec le type de *fichier* s'il s'agit d'un *objet fichier* déjà ouvert ; assurez-vous de ne pas essayer d'écrire une chaîne dans un flux binaire et vice versa.

Modifié dans la version 3.4 : Added the *short_empty_elements* parameter.

Modifié dans la version 3.8 : la méthode `write()` préserve désormais l'ordre des attributs spécifié par l'utilisateur.

Voici le fichier XML à manipuler :

```
<html>
    <head>
        <title>Example page</title>
    </head>
    <body>
        <p>Moved to <a href="http://example.org/">example.org</a>
        or <a href="http://example.com/">example.com</a>.</p>
    </body>
</html>
```

Exemple de modification de l'attribut *target* de chaque lien dans le premier paragraphe :

```
>>> from xml.etree.ElementTree import ElementTree
>>> tree = ElementTree()
>>> tree.parse("index.xhtml")
<Element 'html' at 0xb77e6fac>
>>> p = tree.find("body/p")     # Finds first occurrence of tag p in body
>>> p
<Element 'p' at 0xb77ec26c>
>>> links = list(p.iter("a"))   # Returns list of all links
>>> links
[<Element 'a' at 0xb77ec2ac>, <Element 'a' at 0xb77ec1cc>]
>>> for i in links:             # Iterates through all found links
...     i.attrib["target"] = "blank"
...
>>> tree.write("output.xhtml")
```

Objets *QName*

class xml.etree.ElementTree.**QName**(*text_or_uri*, *tag=None*)

Surcouche à *QName*. Cela peut être utilisé pour envelopper une valeur d'attribut *QName*, afin d'obtenir une gestion appropriée de l'espace de noms en sortie. *text_or_uri* est une chaîne contenant la valeur *QName*, sous la forme *{uri}local* ou, si l'argument *tag* est donné, la partie URI d'un *QName*. Si *tag* est donné, le premier argument est interprété comme un URI et cet argument est interprété comme un nom local. Les instances `QName` sont opaques.

Objets *TreeBuilder*

class xml.etree.ElementTree.**TreeBuilder**(*element_factory=None*, **, comment_factory=None*, *pi_factory=None*, *insert_comments=False*, *insert_pis=False*)

Générateur de structure d'éléments générique. Ce constructeur convertit une séquence d'appels de méthode *start*, *data*, *end*, *comment* et *pi* en une structure d'éléments bien formée. Vous pouvez utiliser cette classe pour créer une structure d'éléments à l'aide d'un analyseur XML personnalisé ou d'un analyseur pour un autre type de format XML.

element_factory, lorsqu'il est donné, doit être un appelable acceptant deux arguments positionnels : une balise et un dictionnaire d'attributs. Elle est censée renvoyer une nouvelle instance d'élément.

Les fonctions *comment_factory* et *pi_factory*, lorsqu'elles sont données, doivent se comporter comme les fonctions `Comment()` et `ProcessingInstruction()` pour créer des commentaires et des instructions de traitement. Lorsqu'elles ne sont pas fournies, les fabriques par défaut sont utilisées. Lorsque *insert_comments* ou *insert_pis* sont vraies, les commentaires et instructions de traitement sont insérés dans l'arborescence s'ils apparaissent dans l'élément racine (mais pas à l'extérieur de celui-ci).

close()

Vide les tampons du générateur et renvoie l'élément de document de niveau supérieur. Renvoie une instance d'`Element`.

data(*data*)

Ajoute du texte à l'élément courant. *data* est une chaîne de caractères. Cela peut être une chaîne d'octets ou une chaîne Unicode.

end(*tag*)

Ferme l'élément courant. *tag* est le nom de l'élément. Renvoie l'élément fermé.

start(*tag*, *attrs*)

Ouvre un nouvel élément. *tag* est le nom de l'élément. *attrs* est un dictionnaire contenant les attributs de l'élément. Renvoie l'élément ouvert.

comment(*text*)

Crée un commentaire avec le *text* donné. Si `insert_comments` est vrai, cela l'ajoute également à l'arborescence.

Ajouté dans la version 3.8.

pi(*target*, *text*)

Creates a process instruction with the given *target* name and *text*. If `insert_pis` is true, this will also add it to the tree.

Ajouté dans la version 3.8.

De plus, un objet `TreeBuilder` personnalisé peut fournir les méthodes suivantes :

doctype(*name*, *pubid*, *system*)

Gère une déclaration *doctype*. *name* est le nom du type de document. *pubid* est l'identifiant public. *system* est l'identifiant du système. Cette méthode n'existe pas sur la classe par défaut `TreeBuilder`.

Ajouté dans la version 3.2.

start_ns(*prefix*, *uri*)

Est appelée chaque fois que l'analyseur rencontre une nouvelle déclaration d'espace de noms, avant le rappel `start()` pour l'élément d'ouverture qui le définit. *prefix* est `''` pour l'espace de noms par défaut et le nom du préfixe d'espace de noms déclaré dans le cas contraire. *uri* est l'URI de l'espace de noms.

Ajouté dans la version 3.8.

end_ns(*prefix*)

Est appelée après le rappel `end()` d'un élément qui a déclaré une correspondance de préfixe d'espace de noms, avec le nom du *prefix* qui passe hors de portée.

Ajouté dans la version 3.8.

class xml.etree.ElementTree.**C14NWriterTarget**(*write*, **, with_comments=False*, *strip_text=False*, *rewrite_prefixes=False*, *qname_aware_tags=None*, *qname_aware_attrs=None*, *exclude_attrs=None*, *exclude_tags=None*)

Écrivain C14N 2.0. Les arguments sont les mêmes que pour la fonction `canonicalize()`. Cette classe ne construit pas d'arborescence mais traduit les événements de rappel directement sous une forme sérialisée à l'aide de la fonction *write*.

Ajouté dans la version 3.8.

Objets *XMLParser*

class xml.etree.ElementTree.**XMLParser**(**, target=None*, *encoding=None*)

Cette classe est l'élément de base du module. Elle utilise `xml.parsers.expat` pour une analyse efficace du XML et fonctionne à base d'événements. Elle peut être alimentée en données XML de manière incrémentielle avec la méthode `feed()`, et les événements d'analyse sont traduits en une API à flux poussé – en appelant les méthodes de rappel de l'objet *target*. Si *target* est omis, le `TreeBuilder` standard est utilisé. Si *encoding*[page 1364, 1] est donné, la valeur remplace l'encodage spécifié dans le fichier XML.

Modifié dans la version 3.8 : Parameters are now *keyword-only*. The *html* argument is no longer supported.

close()

Termine l'envoi des données à l'analyseur. Renvoie le résultat de l'appel de la méthode `close()` de la *target* passée lors de la construction ; par défaut, il s'agit de l'élément de plus haut niveau du document.

feed(*data*)

Fournit des données à l'analyseur. *data* est une donnée encodée.

flush()

Triggers parsing of any previously fed unparsed data, which can be used to ensure more immediate feedback, in particular with Expat >=2.6.0. The implementation of `flush()` temporarily disables reparse deferral with Expat (if currently enabled) and triggers a reparse. Disabling reparse deferral has security consequences ; please see `xml.parsers.expat.xmlparser.SetReparseDeferralEnabled()` for details.

Note that `flush()` has been backported to some prior releases of CPython as a security fix. Check for availability of `flush()` using `hasattr()` if used in code running across a variety of Python versions.

Ajouté dans la version 3.13.

`XMLParser.feed()` appelle la méthode `start(tag, attrs_dict)` de *target* pour chaque balise d'ouverture, sa méthode `end(tag)` pour chaque balise de fermeture, et les données sont traitées par la méthode `data(data)`. Pour d'autres méthodes de rappel prises en charge, consultez la classe `TreeBuilder`. `XMLParser.close()` appelle la méthode `close()` de *target*. `XMLParser` peut être utilisée pour autre chose que seulement construire une structure arborescente. Voici un exemple de comptage de la profondeur maximale d'un fichier XML :

```
>>> from xml.etree.ElementTree import XMLParser
>>> class MaxDepth:                     # The target object of the parser
...     maxDepth = 0
...     depth = 0
...     def start(self, tag, attrib):   # Called for each opening tag.
...         self.depth += 1
...         if self.depth > self.maxDepth:
...             self.maxDepth = self.depth
...     def end(self, tag):             # Called for each closing tag.
...         self.depth -= 1
...     def data(self, data):
...         pass            # We do not need to do anything with data.
...     def close(self):    # Called when all data has been parsed.
...         return self.maxDepth
...
>>> target = MaxDepth()
>>> parser = XMLParser(target=target)
>>> exampleXml = """
... <a>
...   <b>
...   </b>
...   <b>
...     <c>
...       <d>
...       </d>
...     </c>
...   </b>
... </a>"""
>>> parser.feed(exampleXml)
>>> parser.close()
4
```

Objets *XMLPullParser*

class xml.etree.ElementTree.**XMLPullParser**(*events=None*)

Analyseur à flux tiré adapté aux applications non bloquantes. Son API côté entrée est similaire à celle de `XMLParser`, mais au lieu de pousser les appels vers une cible de rappel, `XMLPullParser` collecte une liste interne d'événements d'analyse et permet à l'utilisateur de la lire. *events* est une séquence d'événements à signaler. Les événements pris en charge sont les chaînes `"start"`, `"end"`, `"comment"`, `"pi"`, `"start-ns"` et `"end-ns"` (les événements `"ns"` sont utilisés pour obtenir des informations détaillées sur les espaces de noms). Si *events* est omis, seuls les événements `"end"` sont rapportés.

feed(*data*)

Transmet les données d'octets à l'analyseur.

flush()

Triggers parsing of any previously fed unparsed data, which can be used to ensure more immediate feedback, in particular with Expat >=2.6.0. The implementation of `flush()` temporarily disables reparse deferral with Expat (if currently enabled) and triggers a reparse. Disabling reparse deferral has security consequences ; please see `xml.parsers.expat.xmlparser.SetReparseDeferralEnabled()` for details.

Note that `flush()` has been backported to some prior releases of CPython as a security fix. Check for availability of `flush()` using `hasattr()` if used in code running across a variety of Python versions.

Ajouté dans la version 3.13.

close()

Signale à l'analyseur que le flux de données est terminé. Contrairement à `XMLParser.close()`, cette méthode renvoie toujours `None`. Tous les événements non encore récupérés lorsque l'analyseur est fermé peuvent toujours être lus avec `read_events()`.

read_events()

Renvoie un itérateur sur les événements qui ont été rencontrés dans les données fournies à l'analyseur. L'itérateur génère des paires `(event, elem)`, où *event* est une chaîne représentant le type d'événement (par exemple `"end"`) et *elem* est l'objet `Element` rencontré, ou une autre valeur de contexte comme suit.

— `start`, `end` : l'élément actuel.
— `comment`, `pi` : le commentaire ou l'instruction de traitement actuel
— `start-ns` : couple `(prefix, uri)` nommant la correspondance d'espace de noms déclarée.
— `end-ns` : `None` (cela peut changer dans une version future)

Les événements fournis lors d'un appel précédent à `read_events()` ne seront plus renvoyés. Les événements sont consommés à partir de la file d'attente interne uniquement lorsqu'ils sont récupérés de l'itérateur, donc plusieurs lecteurs itérant en parallèle sur les itérateurs obtenus à partir de `read_events()` auront des résultats imprévisibles.

> **Note**
>
> `XMLPullParser` garantit uniquement qu'il a vu le caractère `">"` d'une balise de début lorsqu'il émet un événement `"start"`, donc les attributs sont définis, mais le contenu des attributs *text* et *tail* n'est pas défini à ce stade. La même chose s'applique aux nœuds enfants de l'élément ; ils peuvent être présents ou non.
>
> Si vous avez besoin d'un élément entièrement rempli, recherchez plutôt les événements `"end"`.

Ajouté dans la version 3.4.

Modifié dans la version 3.8 : les évènements `comment` et `pi` ont été ajoutés.

Exceptions

class xml.etree.ElementTree.**ParseError**

Erreur d'analyse XML, générée par les différentes méthodes d'analyse de ce module lorsque l'analyse échoue. La représentation sous forme de chaîne d'une instance de cette exception contient un message d'erreur convivial. De plus, elle possède les attributs suivants :

code

Code d'erreur numérique de l'analyseur *expat*. Voir la documentation de `xml.parsers.expat` pour la liste des codes d'erreur et leur signification.

position

Un *n*-uplet de numéros de *ligne* et *colonne* indiquant le lieu d'apparition de l'erreur.

Notes

21.6 `xml.dom` --- The Document Object Model API

Code source : Lib/xml/dom/__init__.py

Le Document Object Model, ou "DOM," est une API inter-langage du World Wide Web Consortium (*W3C*) pour accéder et modifier les documents XML. Une implémentation DOM présente le document XML comme un arbre ou autorise le code client à construire une telle structure depuis zéro. Il permet alors d'accéder à la structure à l'aide d'un ensemble d'objet qui fournissent des interfaces bien connues.

Le DOM est extrêmement utile pour les applications à accès aléatoire. SAX ne vous permet de visualiser qu'un seul morceau du document à la fois. Si vous regardez un élément SAX, vous n'avez pas accès à un autre. Si vous regardez un nœud de texte, vous n'avez pas accès à un élément parent. Lorsque vous écrivez une application SAX, vous devez suivre la position de votre programme dans le document quelque part dans votre propre code. SAX ne le fait pas pour vous. De plus, si vous devez examiner un nœud plus lus loin dans le document XML, vous n'avez pas de chance.

Il est tout simplement impossible d'implémenter certains algorithmes avec un modèle évènementiel, sans un accès à un arbre. Bien sûr, vous pourriez construire vous même un arbre à partir des événements SAX mais DOM vous permet d'éviter d'écrire ce code. Le DOM est représentation standard en arbre pour des données XML.

Le DOM (Document Object Model) est défini par le *W3C* en étapes ou *"levels"* (niveaux) selon leur terminologie. Le couplage de l'API de Python est essentiellement basée sur la recommandation DOM Level 2.

Typiquement, les applications DOM commencent par analyser du XML dans du DOM. Comment cela doit être exposé n'est absolument pas décrit par DOM Level 1 et Level 2 ne fournit que des améliorations limités. Il existe une classe `DOMImplementation` qui fournit un accès à des méthodes de création de `Document` mais il n'y a pas de moyen d'accéder à un lecteur/analyseur/constructeur de *document* de façon indépendante de l'implémentation. Il n'est pas également très bien définis comment accéder à ces méthodes sans un objet `Document`. En Python, chaque implémentation fournira une fonction `getDOMImplementation()` . DOM Level 3 ajoute une spécification *Load/Store* (charge/stocke) qui définie une interface pour le lecteur mais qui n'est pas disponible dans la bibliothèque standard de Python.

Une fois que vous avez un objet document DOM, vous pouvez accéder aux parties de votre document XML à travers ses méthodes et propriétés. Ces propriétés sont définis dans les spécifications DOM ; cette portion du manuel de références décrit l'interprétation des ces spécifications en Python.

Les spécifications fournies par le *W3C* définissent les API DOM pour Java, ECMAScript, et OMG IDL. Les correspondances de Python définies ici sont basées pour une grande part sur la version IDL de la spécification mais une conformité stricte n'est pas requise (bien que ces implémentations soient libre d'implémenter le support strict des correspondances de IDL). Voir la section *Conformité* pour une discussion détaillée des pré-requis des correspondances.

Voir aussi

Document Object Model (DOM) Level 2 Specification
La recommandation *W3C* sur laquelle l'API DOM de Python est basée.

Spécification Level 1 Document Object Model (DOM)
La recommandation du *W3C* pour le DOM supporté par `xml.dom.minidom`.

Python Language Mapping Specification
Ceci spécifies les correspondances depuis OMG IDL vers Python.

21.6.1 Contenu du module

Le `xml.dom` contient les fonctions suivantes :

`xml.dom.`**`registerDOMImplementation`**(*name*, *factory*)

Enregistre la fonction *factory* avec le nom *name*. La fonction *factory* doit renvoyer un objet qui implémente l'interface de `DOMImplementation`. La fonction *factory* peut renvoyer le même objet à chaque fois ou un nouveau à chaque appel en accord avec les spécificités de l'implémentation (Par exemple si l'implémentation supporte certaines personnalisations).

`xml.dom.`**`getDOMImplementation`**(*name=None*, *features=()*)

Renvoie une implémentation DOM appropriée. Le *name* est soit connu, soit le nom du module d'une implémentation DOM, soit `None`. Si ce n'est pas `None`, le module correspondant est importé et retourne un objet `DOMImplementation` si l'importation réussit. Si Aucun *name* n'est donné et que la variable d'environnement `PYTHON_DOM` est positionnée, cette variable est utilisée pour trouver l'implémentation.

Si *name* n'est pas donné, la fonction examine les implémentations disponibles pour en trouver une avec l'ensemble des fonctionnalités requises. Si aucune implémentation n'est trouvée, une `ImportError` est levée. La liste de fonctionnalité doit être une séquence de paires `(feature, version)` qui est passée à la méthode `hasFeature()` disponible dans les objets `DOMImplementation`.

Quelques constantes pratiques sont également fournies :

`xml.dom.`**`EMPTY_NAMESPACE`**

La valeur utilisée pour indiquer qu'aucun espace de noms n'est associé à un nœud dans le DOM. Typiquement, ceci est trouvé comme `namespaceURI` dans un nœud ou utilisé comme le paramètre *namespaceURI* dans une méthode spécifique aux espaces de noms.

`xml.dom.`**`XML_NAMESPACE`**

L'URI de l'espace de noms associé avec le préfixe réservé `xml` comme défini par Namespaces in XML (section 4).

`xml.dom.`**`XMLNS_NAMESPACE`**

L'URI de l'espace de noms pour la déclaration des espaces de noms, tel que défini par Document Object Model (DOM) Level 2 Core Specification (section 1.1.8).

`xml.dom.`**`XHTML_NAMESPACE`**

L'URI de l'espace de noms XHTML tel que défini par XHTML 1.0 : The Extensible HyperText Markup Language (section 3.1.1).

Par ailleurs, `xml.dom` contient une classe de base `Node` et les exceptions de DOM. La classe `Node` fournie par ce module n'implémente aucune des méthodes ou des attributs définis par les spécifications DOM ; les implémentations concrètes des DOM doivent fournir les informations suivantes. La classe `Node` fournie part ce modules fournit les constantes utilisées pour l'attribut `nodeType` pour des objets concrets `Node` ; ils sont situés dans les classes plutôt qu'au niveau du module en accord avec les spécifications DOM.

21.6.2 Objets dans le DOM

La documentation finale pour le DOM est la spécification DOM du *W3C*.

Notez que les attributs DOM peuvent également être manipulés comme des nœuds au lieu de simples chaînes. Il est relativement rare que vous ayez besoin de faire cela, cependant, cet usage n'est pas encore documenté.

Interface	Section	Objectif
`DOMImplementation`	*Objets DOMImplementation*	Interface de l'implémentation sous-jacente.
`Node`	*Objets nœuds*	Interface de base pour la majorité des objets dans un document.
`NodeList`	*Objet NodeList*	Interface pour une séquence de nœuds.
`DocumentType`	*Objets DocumnentType*	Informations sur les déclarations nécessaires au traitement d'un document.
`Document`	*Objets Document*	Objet représentant un document entier.
`Element`	*Objets Elements*	Nœuds éléments dans la hiérarchie d'un document.
`Attr`	*Objets Attr*	Valeur des nœuds attributs sur dans des nœuds éléments.
`Comment`	*Objets Comment*	Représentation des commentaires dans le fichier source du document.
`Text`	*Objets Text et CDATASection*	Nœud contenant un contenu texte du document.
`ProcessingInstruct`	*Objets ProcessingInstruction*	Représentation des *Processing Instructions*.

Une Section additionnelle décrit les exceptions définis pour travailler avec le DOM en Python.

Objets DOMImplementation

L'interface `DOMImplementation` fournit un moyen pour les applications de déterminer la disponibilité de fonctionnalités particulières dans le DOM qu'elles utilisent. *DOM Level 2* ajoute la capacité de créer des nouveaux objets `Document` et `DocumentType` utilisant également `DOMImplementation`.

`DOMImplementation.`**`hasFeature`**(*feature*, *version*)

Renvoie `True` si la fonctionnalité identifiée par le couple de chaînes *feature* et *version* est implémentée.

`DOMImplementation.`**`createDocument`**(*namespaceUri*, *qualifiedName*, *doctype*)

Renvoie un nouvel objet `Document` (la racine du DOM), avec un objet fils `Element` ayant les *namespaceUri* et *qualifiedName* passés en paramètre. Le *doctype* doit être un objet `DocumentType` créé par `createDocumentType()` ou `None`. Dans l'API DOM de Python, les deux premiers arguments peuvent également être à `None` de manière à indiquer qu'aucun enfant `Element` ne soit crée.

`DOMImplementation.`**`createDocumentType`**(*qualifiedName*, *publicId*, *systemId*)

Renvoie un nouvel objet `DocumentType` qui encapsule les chaînes *qualifiedName*, *publicId*, et *systemId* passées en paramètre représentant les informations contenues dans dans la déclaration du document XML.

Objets nœuds

Tous les composants d'un document XML sont des sous-classes de `Node`.

`Node.`**`nodeType`**

Un entier représentant le type de nœud. Pour l'objet `Node`, les constantes symboliques pour les types sont `ELEMENT_NODE`, `ATTRIBUTE_NODE`, `TEXT_NODE`, `CDATA_SECTION_NODE`, `ENTITY_NODE`, `PROCESSING_INSTRUCTION_NODE`, `COMMENT_NODE`, `DOCUMENT_NODE`, `DOCUMENT_TYPE_NODE`, `NOTATION_NODE`. Ceci est un attribut en lecture seule.

`Node.`**`parentNode`**

Le parent du nœud courant ou `None` dans le cas du nœud document. La valeur est toujours un objet `Node` ou `None`. Pour les nœuds `Element`, ce sera le parent de l'élément sauf si l'élément est la racine, dans ce cas ce sera l'objet `Document`. Pour les nœuds `Attr`, cela sera toujours `None`. Ceci est un attribut en lecture seule.

`Node.`**`attributes`**

Un objet `NamedNodeMap` d'objet attributs. Seulement les éléments ayant des valeurs seront listés, les autres renverront `None` pour cet attribut. Cet attribut est en lecture seule.

`Node.previousSibling`

Le nœud avec le même parent qui précède immédiatement le nœud courant. Par exemple, l'élément avec la balise fermente qui est juste avant la balise ouvrante de l'élément *self*. Naturellement, les documents XML sont fait de plus que juste des éléments ; donc le *previous sibling* peut être du texte, un commentaire ou autre chose. Si le nœud courant est le premier fils du parent, cet attribut vaudra `None`. Cet attribut est en lecture seule.

`Node.nextSibling`

Le nœud qui suit immédiatement le nœud courant dans le même parent. Voir également `previousSibling`. Si ce nœud est le dernier de son parent, alors l'attribut sera `None`. Cet attribut est en lecture seule.

`Node.childNodes`

Une liste de nœuds contenu dans le nœud courant. Cet attribut est en lecture seule.

`Node.firstChild`

S'il y a des fils, premier fils du nœud courant, sinon `None`. Cet attribut est en lecture seule.

`Node.lastChild`

S'il y a des fils, le dernier nœud fils du nœud courant. Sinon `None`. Cet attribut est en lecture seule.

`Node.localName`

S'il y a un *:*, contient la partie suivante de `tagName` ce *:* sinon la valeur complète de `tagName`. Cette valeur est une chaîne.

`Node.prefix`

La partie de `tagName` précédent le *:* s'il y en a un. sinon une chaîne vide. La valeur est une chaîne ou `None`.

`Node.namespaceURI`

L'espace de noms associé (*namespace* en anglais) au nom de l'élément. Cette valeur est une chaîne ou `None`. Cet attribut est en lecture seule.

`Node.nodeName`

L'attribut a un sens différent pour chaque type de nœud ; se reporter à la spécification DOM pour les détails. Vous obtiendrez toujours l'information que vous obtiendrez à l'aide d'une autre propriété comme `tagName` pour les éléments ou `name` pour les attributs. Pour tous les types de nœuds, la valeur sera soit une chaîne soit `None`. Cet attribut est en lecture seule.

`Node.nodeValue`

L'attribut a un sens différent pour chaque type de nœud ; se reporter à la spécification DOM pour les détails. La situation est similaire à `nodeName`. La valeur est une chaîne ou `None`.

`Node.hasAttributes()`

Renvoie `True` si le nœud a des attributs.

`Node.hasChildNodes()`

Renvoie `True` si le nœud a des nœuds fils.

`Node.isSameNode(other)`

Renvoie `True` si *other* fait référence au même nœud que le nœud courant. Ceci est particulièrement pratique pour une implémentation de DOM qui utilise un ou des mandataires dans son architecture (car plus d'un objet peut se référer au même nœud).

> **Note**
>
> Ceci est basé sur l'API proposé par * DOM Level 3* qui est toujours à l'étape "*working draft*" mais cette interface particulière ne parait pas controversée. Les changement du *W3C* n'affecteront pas nécessairement cette méthode dans l'interface DOM de Python. (bien que toute nouvelle API *W3C* à cet effet soit également supportée).

`Node.appendChild(newChild)`

Ajoute un nouveau nœud fils à ce nœud à la fin de la liste des fils renvoyant *newChild*. Si ce nœud est déjà dans l'arbre, il sera d'abord retiré.

`Node.insertBefore(newChild, refChild)`

Insère un nouveau nœud fils avant un fils existant. Il est impératif que *refChild* soit un fils du nœud, sinon `ValueError` sera levée. *newChild* est renvoyé. Si *refChild* est `None`, *newChild* est inséré à la fin de la liste des fils.

`Node.removeChild(oldChild)`

Retire un nœud fils. *oldChild* doit être un fils de ce nœud ; sinon `ValueError` sera levée. En cas de succès, *oldChild* est renvoyé. Si *oldChild* n'est plus utilisé, sa méthode `unlink()` doit être appelée.

`Node.replaceChild(newChild, oldChild)`

Remplace un nœud existant avec un nouveau. *oldChild* doit être un fils de ce nœud ; sinon `ValueError` sera levée.

`Node.normalize()`

Jointe les nœuds texte adjacents de manière à ce que tous les segments de texte soient stockés dans une seule instance de `Text`. Ceci simplifie le traitement du texte d'un arbre DOM pour de nombreuses applications.

`Node.cloneNode(deep)`

Clone ce nœud. Positionner *deep* signifie que tous les nœuds fils seront également clonés. La méthode renvoi le clone.

Objet NodeList

`NodeList` représente une séquence de nœuds. Ces objets sont utilisés de deux manières dans la recommandation Dom Core : un objet `Element` fournit en fournis liste des nœud fils et les méthodes `getElementsByTagName()` et `getElementsByTagNameNS()` de `Node` renvoient des objet avec cette interface pour représenter les résultats des requêtes.

La recommandation DOM Level 2 définit un attribut et une méthode pour ces objets :

`NodeList.item(i)`

Renvoie le i^e élément de la séquence s'il existe ou `None`. L'index *i* ne peut pas être inférieur à 0 ou supérieur ou égale à la longueur de la séquence.

`NodeList.length`

Le nombre d'éléments dans la séquence.

En plus, l'interface DOM de Python requiert quelques ajouts supplémentaires pour permettre que les objet `NodeList` puissent être utilisés comme des séquences Python. Toutes les implémentations de `NodeList` doivent inclure le support de `__len__()` et de `__getitem__()` ; ceci permet l'itération sur `NodeList` avec l'instruction `for` et un support de la fonction native `len()`.

Si une implémentation de DOM support les modifications du document, l'implémentation de `NodeList` doit également supporter les méthodes `__setitem__()` et `__delitem__()` .

Objets DocumnentType

Les objets de type `DocumentType` fournissent des informations sur les notations et les entités déclarées par un document (incluant les données externes si l'analyseur les utilise et peut les fournir). Le `DocumentType` d'un `Document` est accessible via l'attribut `doctype`. Si le document ne déclare pas de `DOCTYPE`, l'attribut `doctype` vaudra `None` plutôt qu'une instance de cette interface.

`DocumentType` est une spécialisation de `Node` et ajoute les attributs suivants :

`DocumentType.publicId`

L'identifiant publique pour un sous ensemble de la définition type de document (*DTD*). Cela sera une chaîne ou `None`.

`DocumentType.systemId`

L'identifiant système pour un sous ensemble du document de définition type (*DTD*). Cela sera une URI sous la forme d'une chaîne ou `None`.

`DocumentType.internalSubset`

Un chaîne donnant le sous ensemble complet du document. Ceci n'inclut pas les chevrons qui englobe le sous ensemble. Si le document n'a pas de sous ensemble, cela devrait être `None`.

`DocumentType.name`

Le nom de l'élément racine donné dans la déclaration `DOCTYPE` si présente.

`DocumentType.entities`

Ceci est un `NamedNodeMap` donnant les définitions des entités externes. Pour les entités définies plusieurs fois seule la première définition est fournie (les suivantes sont ignorées comme requis par la recommandation XML). Ceci peut retourner `None` si l'information n'est pas fournie à l'analyseur ou si aucune entités n'est définis.

`DocumentType.notations`

Ceci est un `NamedNodeMap` donnant la définition des notations. Pour les notations définies plus d'une fois, seule la première est fournie (les suivante sont ignorées comme requis par la recommandation XML). Ceci peut retourner `None` si l'information n'est pas fournie à l'analyseur ou si aucune entités n'est définis.

Objets Document

Un `Document` représente un document XML en son entier, incluant les éléments qui le constitue, les attributs, les *processing instructions*, commentaires, etc. Rappelez vous qu'il hérite des propriété de `Node`.

`Document.documentElement`

Le seul et unique élément racine du document.

`Document.createElement(tagName)`

Créé et renvoi un nouveau nœud élément. Ce n'est pas inséré dans le document quand il est créé. Vous avez besoin de l'insérer explicitement avec l'une des autres méthodes comme `insertBefore()` ou `appendChild()`.

`Document.createElementNS(namespaceURI, tagName)`

Créé et renvoi un nouvel élément avec un *namespace*. Le *tagName* peut avoir un préfixe. L'élément ne sera pas insérer dans le document quand il est créé. Vous avez besoin de l'insérer explicitement avec l'une des autres méthodes comme `insertBefore()` ou `appendChild()`.

`Document.createTextNode(data)`

Créé et renvoi un nœud texte contenant les *data* passées en paramètre. Comme pour les autres méthodes de création, la méthode n'insère pas le nœud dans l'arbre.

`Document.createComment(data)`

Créé et renvoi un nœud commentaire contenant les *data* passé en commentaire. Comme pour les autres méthodes de création, la méthode n'insère pas le nœud dans l'arbre.

`Document.createProcessingInstruction(target, data)`

Créé et retourne un nœud *processing instruction* contenant les *target* et *data* passés en paramètres. Comme pour les autres méthodes de création, la méthode n'insère pas le nœud dans l'arbre.

`Document.createAttribute(name)`

Créé et renvoi un nœud attribut. Cette méthode n'associe le nœud attribut aucun nœud en particulier. Vous devez utiliser la méthode `setAttributeNode()` sur un objet `Element` approprié pour utiliser une instance d'attribut nouvellement créé.

`Document.createAttributeNS(namespaceURI, qualifiedName)`

Créé et renvoi un nœud attribut avec un *namespace*. Le *tagName* peut avoir un préfixe. Cette méthode n'associe le nœud attribut à aucun nœud en particulier. Vous devez utiliser la méthode `setAttributeNode()` sur un objet `Element` approprié pour utiliser une instance d'attribut nouvellement créé.

`Document.getElementsByTagName(tagName)`

Cherche tout les descendants (fils directs, fils de fils, etc.) avec un nom de balise particulier.

`Document.getElementsByTagNameNS(namespaceURI, localName)`

Cherche tous les descendants (fils directs, fils de fils, etc.) avec un *namespace URI* particulier et un *localName*. Le *localName* fait parti du *namespace* après le préfixe.

Objets Elements

`Element` est une une sous classe de `Node` et donc hérite de tout les éléments de cette classe.

`Element.tagName`

Le nom de l'élément type. Dans un document utilisant des *namespace*, il pourrait y avoir des *:* dedans. La valeur est une chaîne.

`Element.getElementsByTagName(tagName)`

Identique à la méthode équivalente de la classe `Document`.

`Element.getElementsByTagNameNS(namespaceURI, localName)`

Identique à la méthode équivalente de la classe `Document`.

`Element.hasAttribute(name)`

Renvoie `True` si l'élément a un attribut nommé *name*.

`Element.hasAttributeNS(namespaceURI, localName)`

Renvoie `True` si l'élément a un attribut nommé *localName* dans l'espace de noms *namespaceURI*.

`Element.getAttribute(name)`

Retourne la valeur de l'attribut nommé par *name* comme une chaîne. Si un tel attribue n'existe pas, une chaîne vide est retournée comme si l'attribut n'avait aucune valeur.

`Element.getAttributeNode(attrname)`

Retourne le nœud `Attr` pour l'attribut nommé par *attrname*.

`Element.getAttributeNS(namespaceURI, localName)`

Renvoi la valeur de l'attribut nommé par *namespaceURI* et *localName* comme une chaîne. Si un tel attribue n'existe pas, une chaîne vide est retournée comme si l'attribut n'avait aucune valeur.

`Element.getAttributeNodeNS(namespaceURI, localName)`

Renvoi la valeur de l'attribue comme un nœud étant donné *namespaceURI* et *localName*.

`Element.removeAttribute(name)`

Retire un attribut nommé *name*. S'il n'y a aucun attribut correspondant une `NotFoundErr` est levée.

`Element.removeAttributeNode(oldAttr)`

Supprime et renvoi *oldAttr* de la liste des attributs si présent. Si *oldAttr* n'est pas présent, `NotFoundErr` est levée.

`Element.removeAttributeNS(namespaceURI, localName)`

Retire un attribut selon son nom. Notez qu'il utilise un *localName* et nom un *qname*. Aucune exception n'est levée s'il n'y a pas d'attribut correspondant.

`Element.setAttribute(name, value)`

Assigne la valeur à un attribut pour la chaîne.

`Element.setAttributeNode(newAttr)`

Ajoute un nouveau nœud attribut à l'élément, remplaçant un attribut existant si nécessaire si `name` corresponds à un attribut. Si l'attribut en remplace un précédent, l'ancien attribut sera retourné. Si *newAttr* est déjà utilisé, `InuseAttributeErr` sera levée.

`Element.setAttributeNodeNS(newAttr)`

Ajoute un nouveau nœud attribut, remplaçant un attribut existant si `namespaceURI` et `localName` correspondent à un attribut. S'il y a remplacement, l'ancien nœud sera renvoyé. Si *newAttr* est déjà utilisé, `InuseAttributeErr` sera levée.

`Element.setAttributeNS(namespaceURI, qname, value)`

Assigne la valeur d'un attribut depuis une chaîne étant donnée un *namespaceURI* et un *qname*. Notez que *qname* est le nom de l'attribut en entier. Ceci est différent d'au dessus.

Objets Attr

`Attr` hérite `Node` et donc hérite de tout ces attributs.

`Attr.name`

Le nom de l'attribut. Dans un document utilisant des *namespaces*, il pourra inclure un *:*.

`Attr.localName`

La partie du nom suivant le *:* s'il y en a un ou le nom entier sinon. Ceci est un attribut en lecture seule.

`Attr.prefix`

La partie du nom précédent le *:* s'il y en a un ou une chaîne vide.

`Attr.value`

La valeur texte de l'attribut. C'est un synonyme de l'attribut `nodeValue`.

Objets NameNodeMap

`NamedNodeMap` *n'hérite pas* de `Node`.

`NamedNodeMap.length`

La longueur de la liste d'attributs.

`NamedNodeMap.item(index)`

Renvoi un attribut à un index particulier. L'ordre des attribut est arbitraire mais sera constant durant toute la vie du DOM. Chacun des item sera un nœud attribut. Obtenez sa valeur avec `value` de l'attribut.

There are also experimental methods that give this class more mapping behavior. You can use them or you can use the standardized `getAttribute*()` family of methods on the `Element` objects.

Objets Comment

`Comment` représente un commentaire dans le document XML. C'est une sous classe `Node` mais n'a aucune nœuds fils.

`Comment.data`

Le contenu du commentaire comme une chaîne. L'attribut contient tous les caractères entre `<!--` et `-->` mais ne les inclues pas.

Objets Text et CDATASection

L'interface `Text` représente le texte dans un document XML. Si l'analyseur et l'implémentation DOM supporte les extensions XML du DOM, les portion de texte encapsulées dans des section marquées CDATA seront stockées dans des objets `CDATASection`. Ces deux interfaces sont identiques mais fournissent des valeurs différentes pour l'attribut `nodeType`.

Ces interfaces étendent l'interface `Node`. Elles ne peuvent pas avoir de nœuds fils.

`Text.data`

Le contenu du nœud texte comme une chaîne.

> **Note**
>
> L'utilisation d'un nœud `CDATASection` n'indique pas que le nœud représente une section complète marquée CDATA, seulement que le contenu du nœud est le contenu d'une section CDATA. Une seule section CDATA peut représenter plus d'un nœud dans l'arbre du document. Il n'y a aucun moyen de déterminer si deux nœuds `CDATASection` adjacents représentent différentes sections CDATA.

Objets ProcessingInstruction

Représente une *processing instruction* dans un document XML. Hérite de l'interface `Node` et ne peut avoir aucun nœud fils.

`ProcessingInstruction.`**`target`**

Le contenu de la *processing instruction* jusqu'au premier caractère blanc. Cet attribut est en lecture seule.

`ProcessingInstruction.`**`data`**

Le contenu de la *processing instruction* après le premier caractère blanc.

Exceptions

La recommandation *DOM Level 2* définie une seule exception `DOMException` et un nombre de constantes qui permettent aux applications à déterminer quelle type d'erreur s'est produit. Les instances de `DOMException` ont un attribut `code` qui fourni une valeur approprié pour une exception spécifique.

L'interface DOM de Python fournit des constant mais également étends un ensemble d'exception pour qu'il existe une exception spécifique pour chaque code d'exception défini par le DOM. L'implémentation doit lever l'exception spécifique appropriée. Chacune ayant la valeur appropriée pour l'attribut `code`.

`exception` `xml.dom.`**`DOMException`**

Exception de base utilisée pour toutes les exceptions spécifiques du DOM. Cette classe ne peut pas être instanciée directement.

`exception` `xml.dom.`**`DomstringSizeErr`**

Levée quand un intervalle spécifique de texte ne rentre pas dans une chaîne. Cette exception n'est pas réputée être utilisée par les implémentations DOM de Python mais elle peur être levée par des implémentations de DOM qui ne sont pas écrite en Python.

`exception` `xml.dom.`**`HierarchyRequestErr`**

Levée quand l'insertion d'un nœud est tentée dans un type de nœud incompatible.

`exception` `xml.dom.`**`IndexSizeErr`**

Levée quand un index ou la taille d'un paramètre d'une méthode est négatif ou excède les valeurs autorisées.

`exception` `xml.dom.`**`InuseAttributeErr`**

Levée quand l'insertion d'un nœud `Attr` est tenté alors que ce nœud est déjà présent ailleurs dans le document.

`exception` `xml.dom.`**`InvalidAccessErr`**

Levée si un paramètre ou une opération n'est pas supporté par l'objet sous-jacent.

`exception` `xml.dom.`**`InvalidCharacterErr`**

Cette exception est levée quand un paramètre chaîne contient un caractère qui n'est pas autorisé dans le contexte utilisé par la recommandation XML 1.0. Par exemple, lors la tentative de création d'un nœud `Element` avec un espace dans le nom de l'élément.

`exception` `xml.dom.`**`InvalidModificationErr`**

Levée lors de la tentative de modifier le type de nœud.

`exception` `xml.dom.`**`InvalidStateErr`**

Levée quand une tentative est faite d'utiliser un objet non défini ou qui ne sont plus utilisables.

exception xml.dom.**NamespaceErr**

Si une tentative est faite de changer un objet d'une manière qui n'est pas autorisée selon la recommandation Namespaces in XML , cette exception est levée.

exception xml.dom.**NotFoundErr**

Exception quand un nœud n'existe pas dans le contexte référencé. Par exemple, `NamedNodeMap.removeNamedItem()` lèvera cette exception si le nœud passé n'appartient pas à la séquence.

exception xml.dom.**NotSupportedErr**

Levée si l'implémentation ne supporte pas le type d'objet requis ou l'opération.

exception xml.dom.**NoDataAllowedErr**

Levée si la donnée spécifiée pour un nœud n'est pas supportée.

exception xml.dom.**NoModificationAllowedErr**

Levée lors de la tentative de modification sur objet où les modifications ne sont pas autorisées (tels que les nœuds en lecture seule).

exception xml.dom.**SyntaxErr**

Levée quand une chaîne invalide ou illégale est spécifiée.

exception xml.dom.**WrongDocumentErr**

Levée quand un nœud est inséré dans un document différent de celui auquel il appartient et que l'implémentation ne supporte pas la migration d'un document à un autre.

Les codes d'exceptions définis par la recommandation DOM avec leurs correspondances décrites si dessous selon ce tableau :

Constante	Exception
DOMSTRING_SIZE_ERR	*DomstringSizeErr*
HIERARCHY_REQUEST_ERR	*HierarchyRequestErr*
INDEX_SIZE_ERR	*IndexSizeErr*
INUSE_ATTRIBUTE_ERR	*InuseAttributeErr*
INVALID_ACCESS_ERR	*InvalidAccessErr*
INVALID_CHARACTER_ERR	*InvalidCharacterErr*
INVALID_MODIFICATION_ERR	*InvalidModificationErr*
INVALID_STATE_ERR	*InvalidStateErr*
NAMESPACE_ERR	*NamespaceErr*
NOT_FOUND_ERR	*NotFoundErr*
NOT_SUPPORTED_ERR	*NotSupportedErr*
NO_DATA_ALLOWED_ERR	*NoDataAllowedErr*
NO_MODIFICATION_ALLOWED_ERR	*NoModificationAllowedErr*
SYNTAX_ERR	*SyntaxErr*
WRONG_DOCUMENT_ERR	*WrongDocumentErr*

21.6.3 Conformité

Cette section décrit la conformité des pré requis et des relations entre l'API DOM de Python, les recommandations *W3C* DOM et les correspondances OMG IDL pour Python.

Correspondance des types

Les types IDL utilisés dans la spécification DOM correspondent aux types Python selon le tableau suivant.

Type IDL	Type Python
`boolean`	`bool` ou `int`
`int`	`int`
`long int`	`int`
`unsigned int`	`int`
`DOMString`	`str` or `bytes`
`null`	`None`

Méthodes d'accès

Les correspondance de OMG IDL vers Python définissent des fonction d'accès pour les déclarations `attribut` d'IDL à la manière dont Java le fait. Correspondance des déclarations IDL

```
readonly attribute string someValue;
         attribute string anotherValue;
```

donne trois fonctions d'accès : une méthode *get* pour `someValue` (`_get_someValue()`) et des méthodes *get* et *set* pour `anotherValue` (`_get_anotherValue()` et `_set_anotherValue()`). Le *mapping*, en particulier, ne requiert pas que les attributs IDL soient accessible comme des attributs Python normaux : `object.someValue` n'est *pas* requis de fonctionner et peut lever une `AttributeError`.

Cependant, l'API DOM de Python *impose* que les accès par attributs classiques fonctionnent. Par conséquent, les substituts générés par le compilateur IDL de Python ne fonctionneront probablement pas, et des objets façade pourraient être nécessaires côté client si les objets DOM sont manipulés via CORBA. Bien qu'utiliser un client DOM CORBA nécessite une bonne réflexion, les développeurs habitués et expérimentés à l'utilisation de CORBA ne considèrent pas que c'est un problème. Les attributs déclarés `readonly` pourraient ne pas voir leur accès en écriture restreint dans toutes les implémentations du DOM.

Dans l'API DOM de Python, les fonctions d'accès ne sont pas requises. Si elles sont fournies, elles doivent prendre la forme définie par le *mapping* de Python IDL, mais ces méthodes sont considérées inutiles car les attributs sont directement accessible depuis Python. Les fonctions d'accès "Set" ne devraient jamais être fournies pour les attributs `readonly` (en lecture seule).

Les définitions IDL n'embarquent pas entièrement les pré-requis de l'API de DOM API telle que la notion de objets ou que la valeur de retour de `getElementsByTagName()` est dynamique. L'API DOM de Python ne requiert pas des implémentations d'avoir de tel pré-requis.

21.7 `xml.dom.minidom` --- Minimal DOM implementation

Code source : Lib/xml/dom/minidom.py

`xml.dom.minidom` est une implémentation minimale de l'interface *Document Object Model*, avec une API similaire à celle d'autres langages. Elle est censée être plus simple que le DOM complet et également nettement plus petite. Les utilisateurs qui ne maîtrisent pas déjà le DOM devraient plutôt envisager d'utiliser le module `xml.etree.ElementTree` pour leur traitement XML.

> **Avertissement**
>
> le module `xml.dom.minidom` n'est pas sécurisé contre les données construites de façon malveillante. Si vous avez besoin d'analyser des données non sécurisées ou non authentifiées, référez-vous à *Vulnérabilités XML*.

Les applications DOM commencent généralement par analyser du XML dans un DOM. Avec `xml.dom.minidom`, cela se fait via les fonctions d'analyse :

```
from xml.dom.minidom import parse, parseString

dom1 = parse('c:\\temp\\mydata.xml')  # parse an XML file by name

datasource = open('c:\\temp\\mydata.xml')
dom2 = parse(datasource)  # parse an open file

dom3 = parseString('<myxml>Some data<empty/> some more data</myxml>')
```

La fonction `parse()` peut prendre soit un nom de fichier, soit un objet fichier ouvert.

`xml.dom.minidom.`**`parse`**(*filename_or_file*, *parser=None*, *bufsize=None*)

Renvoie un `Document` à partir de l'entrée donnée. *filename_or_file* peut être soit un nom de fichier, soit un objet simili-fichier. *parser*, s'il est donné, doit être un objet analyseur *SAX2*. Cette fonction modifie le pointeur vers le document de l'analyseur et active la prise en charge des espaces de noms ; les autres configurations de l'analyseur (comme la définition d'un résolveur d'entité) doivent avoir été effectuées à l'avance.

Si votre XML se trouve dans une chaîne, vous pouvez utiliser la fonction `parseString()` à la place :

`xml.dom.minidom.`**`parseString`**(*string*, *parser=None*)

Renvoie un `Document` qui représente *string*. Cette méthode crée un objet `io.StringIO` pour la chaîne et le transmet à `parse()`.

Les deux fonctions renvoient un objet `Document` représentant le contenu du document.

Ce que font les fonctions `parse()` et `parseString()`, c'est connecter un analyseur XML avec un « constructeur DOM » qui peut accepter les événements d'analyse de n'importe quel analyseur *SAX* et les convertir en une arborescence DOM. Les noms des fonctions sont peut-être trompeurs, mais sont faciles à comprendre lors de l'apprentissage des interfaces. L'analyse du document sera terminée avant le retour de ces fonctions ; c'est simplement que ces fonctions ne fournissent pas elles-mêmes d'implémentation d'analyseur.

Vous pouvez également créer un `Document` en appelant une méthode sur un objet « DOM Implementation ». Vous pouvez obtenir cet objet soit en appelant la fonction `getDOMImplementation()` du paquet `xml.dom` ou du module `xml.dom.minidom`. Une fois que vous avez un `Document`, vous pouvez y ajouter des nœuds enfants pour remplir le DOM :

```
from xml.dom.minidom import getDOMImplementation

impl = getDOMImplementation()

newdoc = impl.createDocument(None, "some_tag", None)
top_element = newdoc.documentElement
text = newdoc.createTextNode('Some textual content.')
top_element.appendChild(text)
```

Une fois que vous disposez d'un objet document DOM, vous pouvez accéder aux parties de votre document XML via ses propriétés et méthodes. Ces propriétés sont définies dans la spécification DOM. La propriété principale de l'objet document est la propriété `documentElement`. Il vous donne l'élément principal du document XML : celui qui contient tous les autres. Voici un exemple de programme :

```
dom3 = parseString("<myxml>Some data</myxml>")
assert dom3.documentElement.tagName == "myxml"
```

Lorsque vous avez terminé avec une arborescence DOM, vous pouvez éventuellement appeler la méthode `unlink()` pour favoriser un nettoyage précoce des objets désormais inutiles. `unlink()` est une extension spécifique `xml.dom.minidom` de l'API DOM qui rend le nœud et ses descendants essentiellement inutiles. Sinon, le ramasse-miettes de Python finira par s'occuper des objets dans l'arborescence.

Voir aussi

Spécification Level 1 Document Object Model (DOM)
La recommandation du *W3C* pour le DOM pris en charge par `xml.dom.minidom`.

21.7.1 Objets DOM

La définition de l'API DOM pour Python est donnée dans le cadre de la documentation du module `xml.dom`. Cette section répertorie les différences entre l'API et `xml.dom.minidom`.

`Node.unlink()`

Casse les références internes dans le DOM afin qu'elles soient récupérées sur les versions de Python sans ramasse-miettes cyclique. Même lorsque le ramasse-miettes cyclique est disponible, son utilisation peut libérer de grandes quantités de mémoire disponibles plus tôt, donc l'appeler sur les objets DOM dès qu'ils ne sont plus nécessaires est une bonne pratique. Il est suffisant de l'appeler sur l'objet `Document`, mais peut être appelée sur les nœuds enfants pour éliminer les enfants de ce nœud.

Vous pouvez éviter d'appeler cette méthode explicitement en utilisant l'instruction `with`. Le code suivant dissocie automatiquement *dom* lorsque le bloc `with` est quitté :

```
with xml.dom.minidom.parse(datasource) as dom:
    ... # Work with dom.
```

`Node.writexml(writer, indent='', addindent='', newl='', encoding=None, standalone=None)`

Écrit du XML dans l'objet *writer*. Cet objet reçoit du texte mais pas des octets en entrée, il doit avoir une méthode `write()` qui correspond à celle de l'interface objet fichier. Le paramètre *indent* est l'indentation du nœud actuel. Le paramètre *addindent* est l'indentation incrémentielle à utiliser pour les sous-nœuds du nœud actuel. Le paramètre *newl* spécifie la chaîne à utiliser pour terminer les nouvelles lignes.

Pour le nœud `Document`, un argument nommé supplémentaire *encoding* peut être utilisé pour spécifier le champ d'encodage de l'en-tête XML.

De même, l'indication explicite de l'argument *standalone* entraîne l'ajout des déclarations de document autonome au prologue du document XML. Si la valeur est définie sur `True`, `standalone="yes"` est ajoutée, sinon elle est définie sur `"no"`. Par défaut la déclaration n'est pas écrite dans le document.

Modifié dans la version 3.8 : la méthode `writexml()` préserve désormais l'ordre des attributs spécifié par l'utilisateur.

Modifié dans la version 3.9 : le paramètre *standalone* a été ajouté.

`Node.toxml(encoding=None, standalone=None)`

Renvoie une chaîne ou une chaîne d'octets contenant le XML représenté par le nœud DOM.

Avec un argument explicite *encoding*[1], le résultat est une chaîne d'octets dans l'encodage spécifié. Sans argument *encoding*, le résultat est une chaîne Unicode et la déclaration XML dans la chaîne résultante ne spécifie pas d'encodage. Encoder cette chaîne dans un codage autre que UTF-8 est probablement incorrect, puisque UTF-8 est l'encodage par défaut de XML.

L'argument *standalone* se comporte exactement comme dans `writexml()`.

Modifié dans la version 3.8 : la méthode `toxml()` préserve désormais l'ordre des attributs spécifié par l'utilisateur.

Modifié dans la version 3.9 : le paramètre *standalone* a été ajouté.

`Node.toprettyxml(indent='\t', newl='\n', encoding=None, standalone=None)`

Renvoie une version du document agréablement mise en forme. *indent* spécifie la chaîne d'indentation et est par défaut une tabulation ; *newl* spécifie la chaîne émise à la fin de chaque ligne et la valeur par défaut est `\n`.

1. Le nom de codage inclus dans la sortie XML doit être conforme aux normes appropriées. Par exemple, `"UTF-8"` est valide, mais `"UTF8"` ne l'est pas dans la déclaration d'un document XML, même si Python l'accepte comme nom de codage. Voir https://www.w3.org/TR/2006/REC-xml11-20060816/#NT-EncodingDecl et https://www.iana.org/assignments/character-sets/character-sets.xhtml.

L'argument *encoding* se comporte comme l'argument correspondant de `toxml()`.

L'argument *standalone* se comporte exactement comme dans `writexml()`.

Modifié dans la version 3.8 : la méthode `toprettyxml()` préserve désormais l'ordre des attributs spécifié par l'utilisateur.

Modifié dans la version 3.9 : le paramètre *standalone* a été ajouté.

21.7.2 Exemple DOM

Cet exemple de programme est un exemple assez réaliste de programme simple. Dans ce cas particulier, nous ne profitons pas beaucoup de la flexibilité du DOM.

```
import xml.dom.minidom

document = """\
<slideshow>
<title>Demo slideshow</title>
<slide><title>Slide title</title>
<point>This is a demo</point>
<point>Of a program for processing slides</point>
</slide>

<slide><title>Another demo slide</title>
<point>It is important</point>
<point>To have more than</point>
<point>one slide</point>
</slide>
</slideshow>
"""

dom = xml.dom.minidom.parseString(document)

def getText(nodelist):
    rc = []
    for node in nodelist:
        if node.nodeType == node.TEXT_NODE:
            rc.append(node.data)
    return ''.join(rc)

def handleSlideshow(slideshow):
    print("<html>")
    handleSlideshowTitle(slideshow.getElementsByTagName("title")[0])
    slides = slideshow.getElementsByTagName("slide")
    handleToc(slides)
    handleSlides(slides)
    print("</html>")

def handleSlides(slides):
    for slide in slides:
        handleSlide(slide)

def handleSlide(slide):
    handleSlideTitle(slide.getElementsByTagName("title")[0])
    handlePoints(slide.getElementsByTagName("point"))

def handleSlideshowTitle(title):
    print(f"<title>{getText(title.childNodes)}</title>")
```

(suite sur la page suivante)

(suite de la page précédente)

```
def handleSlideTitle(title):
    print(f"<h2>{getText(title.childNodes)}</h2>")

def handlePoints(points):
    print("<ul>")
    for point in points:
        handlePoint(point)
    print("</ul>")

def handlePoint(point):
    print(f"<li>{getText(point.childNodes)}</li>")

def handleToc(slides):
    for slide in slides:
        title = slide.getElementsByTagName("title")[0]
        print(f"<p>{getText(title.childNodes)}</p>")

handleSlideshow(dom)
```

21.7.3 *minidom* et le standard DOM

Le module `xml.dom.minidom` est essentiellement une interface vers DOM compatible DOM 1.0 avec certaines fonctionnalités DOM 2 (principalement des fonctionnalités d'espace de noms).

L'utilisation de l'interface DOM en Python est simple. Les règles de correspondances suivantes s'appliquent :

- Les interfaces sont accessibles via des objets d'instance. Les applications ne doivent pas instancier les classes elles-mêmes ; elles doivent utiliser les fonctions de création disponibles sur l'objet `Document`. Les interfaces dérivées prennent en charge toutes les opérations (et attributs) des interfaces de base, ainsi que toutes les nouvelles opérations.
- Les opérations sont utilisées comme méthodes. Puisque le DOM utilise uniquement les paramètres `in`, les arguments sont passés dans l'ordre normal (de gauche à droite). Il n'y a pas d'argument facultatif. Les opérations `void` renvoient `None`.
- Les attributs *IDL* (*Interface Description Language*) correspondent aux attributs d'instance. Pour des raisons de compatibilité avec la correspondance OMG (*Object Management Group*) du langage *IDL* pour Python, un attribut `foo` est également accessible via les méthodes d'accès `_get_foo()` et `_set_foo()`. Les attributs `readonly` ne doivent pas être modifiés ; ce n'est pas vérifié au moment de l'exécution.
- Les types `short int`, `unsigned int`, `unsigned long long` et `boolean` correspondent tous à des objets entiers Python.
- Le type `DOMString` correspond aux chaînes Python. `xml.dom.minidom` prend en charge soit les octets, soit les chaînes, et produit normalement des chaînes. Les valeurs de type `DOMString` peuvent également être `None` là où la valeur *IDL* `null` est autorisée par la spécification DOM du W3C.
- Les déclarations `const` correspondent aux variables dans leur portée respective (par exemple `xml.dom.minidom.Node.PROCESSING_INSTRUCTION_NODE`) ; elles ne doivent pas être modifiées.
- `DOMException` n'est actuellement pas pris en charge dans `xml.dom.minidom`. En lieu et place, `xml.dom.minidom` utilise des exceptions Python standard telles que `TypeError` et `AttributeError`.
- Les objets `NodeList` sont implémentés à l'aide du type de liste natif de Python. Ces objets fournissent l'interface définie dans la spécification DOM, mais avec les versions antérieures de Python, ils ne prennent pas en charge l'API officielle. C'est cependant bien plus « Pythonique » que l'interface définie dans les recommandations du W3C.

Les interfaces suivantes n'ont aucune implémentation dans `xml.dom.minidom` :

- `DOMTimeStamp`
- `EntityReference`

La plupart d'entre elles reflètent des informations contenues dans le document XML qui ne sont pas d'utilité générale pour la plupart des utilisateurs du DOM.

Notes

21.8 xml.dom.pulldom --- Support for building partial DOM trees

Code source : Lib/xml/dom/pulldom.py

Le module *xml.dom.pulldom* fournit un « analyseur à flux tiré » auquel on peut également demander de produire des fragments du document accessibles via le DOM si nécessaire. Le concept de base consiste à extraire des « événements » d'un flux XML entrant et à les traiter. Contrairement à *SAX* qui utilise également un modèle de traitement basé sur les événements avec des rappels, l'utilisateur d'un analyseur à flux tiré est responsable de l'extraction explicite des événements du flux, en parcourant ces événements en boucle jusqu'à ce que le traitement soit terminé ou qu'une condition d'erreur se produise.

> **Avertissement**
>
> le module *xml.dom.pulldom* n'est pas sécurisé contre les données construites de façon malveillante. Si vous avez besoin d'analyser des données non sécurisées ou non authentifiées, référez-vous à *Vulnérabilités XML*.

Modifié dans la version 3.7.1 : l'analyseur *SAX* ne traite plus les entités externes générales par défaut pour augmenter la sécurité par défaut. Pour activer le traitement des entités externes, transmettez une instance d'analyseur personnalisée à :

```
from xml.dom.pulldom import parse
from xml.sax import make_parser
from xml.sax.handler import feature_external_ges

parser = make_parser()
parser.setFeature(feature_external_ges, True)
parse(filename, parser=parser)
```

Exemple :

```
from xml.dom import pulldom

doc = pulldom.parse('sales_items.xml')
for event, node in doc:
    if event == pulldom.START_ELEMENT and node.tagName == 'item':
        if int(node.getAttribute('price')) > 50:
            doc.expandNode(node)
            print(node.toxml())
```

`event` est une constante parmi les éléments suivants :

- `START_ELEMENT`
- `END_ELEMENT`
- `COMMENT`
- `START_DOCUMENT`
- `END_DOCUMENT`
- `CHARACTERS`
- `PROCESSING_INSTRUCTION`
- `IGNORABLE_WHITESPACE`

`node` est un objet de type `xml.dom.minidom.Document`, `xml.dom.minidom.Element` ou `xml.dom.minidom.Text`.

Puisque le document est traité comme un flux « plat » d'événements, « l'arborescence » du document est implicitement parcourue et les éléments souhaités sont trouvés quelle que soit leur profondeur dans l'arborescence. En d'autres termes, il n'est pas nécessaire de prendre en compte les problèmes hiérarchiques tels que la recherche récursive des nœuds du document, même dans le cas où le contexte des éléments est important, et où il faudrait soit maintenir un

état lié au contexte (c'est-à-dire se souvenir où l'on se trouve dans le document à un moment donné), soit utiliser la méthode `DOMEventStream.expandNode()` et passer au traitement lié du DOM.

class xml.dom.pulldom.**PullDom**(*documentFactory=None*)

Sous-classe de `xml.sax.handler.ContentHandler`.

class xml.dom.pulldom.**SAX2DOM**(*documentFactory=None*)

Sous-classe de `xml.sax.handler.ContentHandler`.

xml.dom.pulldom.**parse**(*stream_or_string*, *parser=None*, *bufsize=None*)

Renvoie un `DOMEventStream` à partir de l'entrée donnée. *stream_or_string* peut être soit un nom de fichier, soit un objet simili-fichier. *parser*, s'il est donné, doit être un objet `XMLReader`. Cette fonction modifie le gestionnaire de documents de l'analyseur et active la prise en charge des espaces de noms ; les autres configurations de l'analyseur (comme la définition d'un résolveur d'entité) doivent avoir été effectuées à l'avance.

Si vous avez du XML dans une chaîne, vous pouvez utiliser la fonction `parseString()` à la place :

xml.dom.pulldom.**parseString**(*string*, *parser=None*)

Renvoie un `DOMEventStream` qui représente *string* (Unicode).

xml.dom.pulldom.**default_bufsize**

Valeur par défaut du paramètre *bufsize* de `parse()`.

La valeur de cette variable peut être modifiée avant d'appeler `parse()` et la nouvelle valeur prendra effet.

21.8.1 Objets *DOMEventStream*

class xml.dom.pulldom.**DOMEventStream**(*stream*, *parser*, *bufsize*)

Modifié dans la version 3.11 : Support for `__getitem__()` method has been removed.

getEvent()

Renvoie un *n*-uplet contenant *event* et le *node* actuel sous la forme d'un `xml.dom.minidom.Document` si l'événement est égal à `START_DOCUMENT`, un `xml.dom.minidom.Element` si l'événement est égal à `START_ELEMENT` ou `END_ELEMENT`, ou un `xml.dom.minidom.Text` si l'événement est égal à `CHARACTERS`. Le nœud actuel ne contient pas d'informations sur ses enfants, sauf si `expandNode()` est appelé.

expandNode(*node*)

Développe tous les enfants de *node* dans *node*. Par exemple :

```
from xml.dom import pulldom

xml = '<html><title>Foo</title> <p>Some text <div>and more</div></p> </
↪html>'
doc = pulldom.parseString(xml)
for event, node in doc:
    if event == pulldom.START_ELEMENT and node.tagName == 'p':
        # Following statement only prints '<p/>'
        print(node.toxml())
        doc.expandNode(node)
        # Following statement prints node with all its children '<p>Some␣
↪text <div>and more</div></p>'
        print(node.toxml())
```

reset()

21.9 `xml.sax` --- Support for SAX2 parsers

Code source : Lib/xml/sax/__init__.py

Le paquet `xml.sax` fournit des modules qui implémentent l'interface *Simple API for XML (SAX)* pour Python. Le paquet en lui-même fournit les exceptions *SAX* et les fonctions les plus utiles qui sont le plus utilisées par les utilisateurs de l'API *SAX*.

> **Avertissement**
>
> le module `xml.sax` n'est pas sécurisé contre les données construites de façon malveillante. Si vous avez besoin d'analyser des données non sécurisées ou non authentifiées, lisez *Vulnérabilités XML*.

Modifié dans la version 3.7.1 : l'analyseur *SAX* ne traite plus les entités externes générales par défaut pour augmenter la sécurité. Auparavant, l'analyseur créait des connexions réseau pour extraire des fichiers distants ou des fichiers locaux chargés à partir du système de fichiers pour les *DTD* et les entités. La fonctionnalité peut être activée à nouveau avec la méthode `setFeature()` sur l'objet analyseur et l'argument `feature_external_ges`.

Les fonctions les plus utiles sont :

`xml.sax.`**`make_parser`**(*parser_list=[]*)

Crée et renvoie un objet `XMLReader` *SAX*. Le premier analyseur trouvé est utilisé. Si *parser_list* est fourni, il doit être un itérable de chaînes de caractères qui nomme des modules qui ont une fonction nommée `create_parser()`. Les modules listés dans *parser_list* sont utilisés avant les modules dans la liste par défaut des analyseurs.

Modifié dans la version 3.8 : l'argument *parser_list* peut être n'importe quel itérable, pas seulement une liste.

`xml.sax.`**`parse`**(*filename_or_stream*, *handler*, *error_handler=handler.ErrorHandler()*)

Crée un analyseur *SAX* et l'utilise pour analyser un document. Le document transmis comme *filename_or_stream*, peut être un nom de fichier ou un objet fichier. Le paramètre *handler* doit être une instance `ContentHandler` *SAX*. Si un *error_handler* est donné, il doit être un `ErrorHandler` *SAX* ; s'il est omis, une `SAXParseException` est levée sur toutes les erreurs. Il n'y a pas de valeur de retour, tout le travail doit être fait par le *handler* transmis.

`xml.sax.`**`parseString`**(*string*, *handler*, *error_handler=handler.ErrorHandler()*)

Similaire à `parse()`, mais analyse à partir d'un espace mémoire *string* reçu en tant que paramètre. *string* doit être une instance `str` ou un objet *octet-compatible*.

Modifié dans la version 3.5 : ajout de la gestion des instances `str`.

Une application *SAX* typique utilise trois types d'objets : les *readers*, les *handlers* et les sources d'entrée. *Reader* dans ce contexte est un autre terme pour « analyseur », c'est-à-dire un morceau de code qui lit les octets ou les caractères de la source d'entrée et qui produit une séquence d'événements. Les événements sont ensuite distribués aux objets du *handler*, c'est-à-dire que le lecteur appelle une méthode sur le *handler*. L'application doit donc obtenir un objet *reader*, créer ou ouvrir les sources d'entrée, créer les *handlers* et connecter ces objets tous ensemble. La dernière étape de la préparation, le *reader* est appelé à analyser l'entrée. Pendant l'analyse, les méthodes sur les objets du *handler* sont appelées en fonction d'événements structurels et syntaxiques à partir des données d'entrée.

Pour ces objets, seules les interfaces sont pertinentes ; elles ne sont pas normalement instanciées par l'application elle-même. Puisque Python n'a pas de notion explicite d'interface, elles sont formellement introduites en tant que classes, mais les applications peuvent utiliser des implémentations qui n'héritent pas des classes fournies. Les interfaces `InputSource`, `Locator`, `Attributes`, `AttributesNS` et `XMLReader` sont définies dans le module `xml.sax.xmlreader`. Les interfaces du *handler* sont définies dans `xml.sax.handler`. Pour plus de commodité, `xml.sax.xmlreader.InputSource` (qui est souvent instanciée directement) et les classes du *handler* sont également disponibles à partir de `xml.sax`. Ces interfaces sont décrites ci-dessous.

En plus de ces classes, `xml.sax` fournit les classes d'exceptions suivantes.

exception xml.sax.**SAXException**(*msg*, *exception=None*)

Encapsule une erreur ou un avertissement XML. Cette classe peut contenir une erreur de base ou une information d'avertissement soit de l'analyseur XML ou de l'application : elle peut être sous-classée pour fournir des fonctionnalités supplémentaires ou pour ajouter une localisation. Noter que même si les *handlers* définis dans l'interface `ErrorHandler` reçoivent des instances de cette exception, ce n'est pas nécessaire de lever l'exception --- il est également utile en tant que conteneur pour l'information.

Quand elle est instanciée, *msg* doit être une description intelligible de l'erreur. Le paramètre optionnel *exception*, s'il est donné, doit être `None` ou une exception qui a été interceptée par le code d'analyse et qui est transmise comme information.

Ceci est la classe de base pour les autres classes d'exception *SAX*.

exception xml.sax.**SAXParseException**(*msg*, *exception*, *locator*)

Sous-classe de `SAXException` élevée sur les erreurs d'analyse. Les instances de cette classe sont passées aux méthodes de l'interface `ErrorHandler` *SAX* pour fournir des informations sur l'erreur d'analyse. Cette classe prend aussi en charge l'interface *SAX* `Locator` comme l'interface `SAXException`.

exception xml.sax.**SAXNotRecognizedException**(*msg*, *exception=None*)

Sous-classe de `SAXException` levée quand un `XMLReader` *SAX* est confronté à une caractéristique ou à une propriété non reconnue. Les applications et les extensions *SAX* peuvent utiliser cette classe à des fins similaires.

exception xml.sax.**SAXNotSupportedException**(*msg*, *exception=None*)

Sous-classe de `SAXException` levée quand un `XMLReader` *SAX* est demandé pour activer une fonctionnalité qui n'est pas prise en charge, ou pour définir une propriété à une valeur que l'implémentation ne prend pas en charge. Les applications et les extensions *SAX* peuvent utiliser cette classe à des fins similaires.

Voir aussi

SAX : l'API simple pour XML
Ce site est le point focal pour la définition de l'API *SAX*. Il offre une implémentation Java et une documentation en ligne. Des liens pour l'implémentation et des informations historiques sont également disponibles.

Module `xml.sax.handler`
Définitions des interfaces pour les objets fournis par l'application.

Module `xml.sax.saxutils`
Fonctions pratiques pour une utilisation dans les applications *SAX*.

Module `xml.sax.xmlreader`
Définitions des interfaces pour les objets fournis par l'analyseur.

21.9.1 Les objets *SAXException*

La classe d'exception `SAXException` gère les méthodes suivantes :

SAXException.**getMessage**()

Renvoie un message intelligible décrivant la condition d'erreur.

SAXException.**getException**()

Renvoie un objet d'exception encapsulé ou `None`.

21.10 `xml.sax.handler` --- Base classes for SAX handlers

Code source : Lib/xml/sax/handler.py

L'API *SAX* définit cinq types de gestionnaires : les gestionnaires de contenu, les gestionnaires de *DTD*, les gestionnaires d'erreurs, les résolveurs d'entités et les gestionnaires lexicaux. Les applications n'ont normalement besoin d'implémenter que les interfaces dont les événements les intéressent ; elles peuvent implémenter les interfaces dans un seul

objet ou dans plusieurs objets. Les implémentations de gestionnaires doivent hériter des classes mères fournies dans le module `xml.sax.handler`, afin que toutes les méthodes aient des implémentations par défaut.

class `xml.sax.handler.`**`ContentHandler`**

Il s'agit de l'interface de rappel principale de *SAX* et de la plus importante pour les applications. L'ordre des événements dans cette interface reflète l'ordre des informations dans le document.

class `xml.sax.handler.`**`DTDHandler`**

Gestionnaires d'évènements *DTD*

Cette interface concerne uniquement les événements *DTD* requis pour une analyse basique (entités et attributs non analysés).

class `xml.sax.handler.`**`EntityResolver`**

Interface de base pour résoudre les entités. Si vous créez un objet implémentant cette interface, puis enregistrez l'objet auprès de votre analyseur, l'analyseur appellera la méthode de votre objet pour résoudre toutes les entités externes.

class `xml.sax.handler.`**`ErrorHandler`**

Interface utilisée par l'analyseur pour présenter des messages d'erreur et d'avertissement à l'application. Les méthodes de cet objet permettent de choisir si les erreurs doivent être immédiatement converties en exceptions ou si elles doivent être traitées d'une autre manière.

class `xml.sax.handler.`**`LexicalHandler`**

Interface utilisée par l'analyseur pour représenter des événements rares qui n'intéressent que peu d'applications.

En plus de ces classes, `xml.sax.handler` fournit des constantes symboliques pour les noms de fonctionnalités et de propriétés.

`xml.sax.handler.`**`feature_namespaces`**

valeur : `"http://xml.org/sax/features/namespaces"`
vrai : effectue le traitement de l'espace de noms.
faux : n'effectue pas le traitement des espaces de noms (implique *namespace-prefixes* ; valeur par défaut).
accès : (lors de l'analyse) en lecture seule ; (hors de l'analyse) lecture-écriture

`xml.sax.handler.`**`feature_namespace_prefixes`**

valeur : `"http://xml.org/sax/features/namespace-prefixes"`
vrai : renvoie les noms et attributs avec les préfixes originaux utilisés dans les déclarations d'espaces de noms.
faux : ne renvoie pas les attributs utilisés pour les déclarations d'espaces de noms et, éventuellement, ne signale pas les noms avec les préfixes originaux (par défaut).
accès : (lors de l'analyse) en lecture seule ; (hors de l'analyse) lecture-écriture

`xml.sax.handler.`**`feature_string_interning`**

valeur : `"http://xml.org/sax/features/string-interning"`
vrai : tous les noms d'éléments, préfixes, noms d'attributs, URI d'espace de noms et noms locaux sont rendus internes à l'aide de la fonction native correspondante.
faux : les noms ne sont pas nécessairement rendus internes, bien qu'ils puissent l'être (par défaut).
accès : (lors de l'analyse) en lecture seule ; (hors de l'analyse) lecture-écriture

`xml.sax.handler.`**`feature_validation`**

valeur : `"http://xml.org/sax/features/validation"`
vrai : signale toutes les erreurs de validation (implique *external-general-entities* et *external-parameter-entities*).
faux : ne signale pas les erreurs de validation.
accès : (lors de l'analyse) en lecture seule ; (hors de l'analyse) lecture-écriture

`xml.sax.handler.`**`feature_external_ges`**

valeur : `"http://xml.org/sax/features/external-general-entities"`

vrai : Inclut toutes les entités générales (texte) externes.
faux : n'inclut pas les entités générales externes.
accès : (lors de l'analyse) en lecture seule ; (hors de l'analyse) lecture-écriture

`xml.sax.handler.`**`feature_external_pes`**

valeur : `"http://xml.org/sax/features/external-parameter-entities"`
vrai : Inclut toutes les entités de paramètres externes, y compris les sous-ensembles *DTD* externes.
faux : n'inclue aucune entité de paramètre externe, même les sous-ensembles *DTD* externes.
accès : (lors de l'analyse) en lecture seule ; (hors de l'analyse) lecture-écriture

`xml.sax.handler.`**`all_features`**
Liste de toutes les fonctionnalités.

`xml.sax.handler.`**`property_lexical_handler`**

valeur : `"http://xml.org/sax/properties/lexical-handler"`
type de données : `xml.sax.handler.LexicalHandler` (non pris en charge dans Python 2)
description : gestionnaire d'extension facultatif pour les événements lexicaux tels que les commentaires.
accès : lecture-écriture

`xml.sax.handler.`**`property_declaration_handler`**

valeur : `"http://xml.org/sax/properties/declaration-handler"`
type de données : xml.sax.sax2lib.DeclHandler (non pris en charge dans Python 2)
description : gestionnaire d'extension facultatif pour les événements liés à la *DTD* autres que les notations et les entités non analysées.
accès : lecture-écriture

`xml.sax.handler.`**`property_dom_node`**

valeur : `"http://xml.org/sax/properties/dom-node"`
type de données : org.w3c.dom.Node (non pris en charge dans Python 2)
description : lors de l'analyse, le nœud DOM actuel visité s'il s'agit d'un itérateur DOM ; hors de l'analyse, le nœud racine du DOM pour l'itération.
accès : (lors de l'analyse) en lecture seule ; (hors de l'analyse) lecture-écriture

`xml.sax.handler.`**`property_xml_string`**

valeur : `"http://xml.org/sax/properties/xml-string"`
type de données : octets
description : chaîne littérale de caractères qui était la source de l'événement en cours.
accès : lecture seule

`xml.sax.handler.`**`all_properties`**
Liste de tous les noms de propriétés connus.

21.10.1 Objets *ContentHandler*

Les utilisateurs doivent sous-classer `ContentHandler` dans leur application. Les méthodes suivantes sont appelées par l'analyseur pour notifier les événements appropriés dans le document d'entrée :

`ContentHandler.`**`setDocumentLocator`**(*locator*)

Appelé par l'analyseur pour donner à l'application un localisateur permettant de localiser l'origine des événements du document.

Les analyseurs *SAX* sont fortement encouragés (bien que ce ne soit pas absolument obligatoire) à fournir un localisateur : s'ils le font, ils doivent fournir le localisateur à l'application en appelant cette méthode avant d'invoquer l'une des autres méthodes de l'interface *DocumentHandler*.

Le localisateur permet à l'application de déterminer la position finale de tout événement lié au document, même si l'analyseur ne signale pas d'erreur. En règle générale, l'application utilise ces informations pour signaler

ses propres erreurs (telles que le contenu d'un champ texte qui ne correspond pas aux règles métier d'une application). Les informations renvoyées par le localisateur ne sont probablement pas suffisantes pour être utilisées avec un moteur de recherche.

Notez que le localisateur renvoie des informations correctes pendant la durée de l'invocation des événements dans cette interface. L'application ne doit pas tenter de l'utiliser à un autre moment.

`ContentHandler.`**`startDocument`**`()`

Reçoit la notification correspondant au début du document.

L'analyseur *SAX* invoque cette méthode une seule fois, avant toute autre méthode dans cette interface ou dans *DTDHandler* (sauf pour `setDocumentLocator()`).

`ContentHandler.`**`endDocument`**`()`

Reçoit la notification correspondant à la fin du document.

L'analyseur *SAX* n'invoque cette méthode qu'une seule fois et c'est la dernière méthode invoquée lors de l'analyse. L'analyseur ne doit pas invoquer cette méthode tant qu'il n'a pas abandonné l'analyse (en raison d'une erreur irrécupérable) ou atteint la fin de l'entrée.

`ContentHandler.`**`startPrefixMapping`**`(`*prefix*, *uri*`)`

Commence la portée d'une correspondance d'espace de noms préfixe-URI.

Les informations de cet événement ne sont pas nécessaires au traitement normal des espaces de noms : le lecteur XML *SAX* remplace automatiquement les préfixes des noms d'éléments et d'attributs lorsque la fonctionnalité `feature_namespaces` est activée (la valeur par défaut).

Il existe cependant des cas où les applications doivent utiliser des préfixes dans des champs textes ou dans les valeurs d'attribut, où ils ne peuvent pas être développés automatiquement en toute sécurité ; les événements `startPrefixMapping()` et `endPrefixMapping()` fournissent les informations à l'application pour développer elle-même les préfixes dans ces contextes, si nécessaire.

Notez qu'il n'est pas garanti que les événements `startPrefixMapping()` et `endPrefixMapping()` soient correctement imbriqués les uns par rapport aux autres : tout événement `startPrefixMapping()` se produit avant l'événement `startElement()` correspondant, et tout événement `endPrefixMapping()` se produit après l'événement `endElement()` correspondant, mais leur ordre n'est pas garanti.

`ContentHandler.`**`endPrefixMapping`**`(`*prefix*`)`

Termine la portée d'une correspondance préfixe-URI.

Voir `startPrefixMapping()` pour plus de détails. Cet événement se produit toujours après l'événement `endElement()` correspondant, mais l'ordre des événements `endPrefixMapping()` n'est pas garanti entre eux.

`ContentHandler.`**`startElement`**`(`*name*, *attrs*`)`

Signale le début d'un élément en mode sans espaces de noms.

Le paramètre *name* contient le nom brut XML 1.0 du type d'élément sous forme de chaîne et le paramètre *attrs* contient un objet de l'interface `Attributes` (voir *The Attributes Interface*) contenant les attributs de l'élément. L'objet passé comme *attrs* peut être réutilisé par l'analyseur ; conserver une référence à celui-ci n'est pas un moyen fiable de conserver une copie des attributs. Pour conserver une copie des attributs, utilisez la méthode `copy()` de l'objet *attrs*.

`ContentHandler.`**`endElement`**`(`*name*`)`

Signale la fin d'un élément en mode sans espaces de noms.

Le paramètre *name* contient le nom du type d'élément, tout comme pour l'événement `startElement()`.

`ContentHandler.`**`startElementNS`**`(`*name*, *qname*, *attrs*`)`

Signale le début d'un élément en mode avec espaces de noms.

Le paramètre *name* contient le nom du type d'élément sous forme de *n*-uplet `(uri, localname)`, le paramètre *qname* contient le nom XML 1.0 brut utilisé dans le document source et le paramètre *attrs* contient une instance de l'interface `AttributesNS` (voir *The AttributesNS Interface*) contenant les attributs de l'élément. Si aucun espace de noms n'est associé à l'élément, le composant *uri* de *name* est `None`. L'objet passé comme *attrs*

peut être réutilisé par l'analyseur ; conserver une référence à celui-ci n'est pas un moyen fiable de conserver une copie des attributs. Pour conserver une copie des attributs, utilisez la méthode `copy()` de l'objet *attrs*.

Les analyseurs peuvent définir le paramètre *qname* à `None`, à moins que la fonctionnalité `feature_namespace_prefixes` ne soit activée.

`ContentHandler.`**`endElementNS`**(*name*, *qname*)

Signale la fin d'un élément en mode avec espaces de noms.

Le paramètre *name* contient le nom du type d'élément, tout comme avec la méthode `startElementNS()`, de même que le paramètre *qname*.

`ContentHandler.`**`characters`**(*content*)

Reçoit une notification correspondant à des données de type caractère.

L'analyseur appelle cette méthode pour signaler chaque morceau de données de caractères. Les analyseurs *SAX* peuvent renvoyer toutes les données de caractères contiguës en un seul morceau ou ils peuvent les diviser en plusieurs morceaux ; cependant, tous les caractères d'un événement doivent provenir de la même entité externe afin que le localisateur fournisse des informations utiles.

content peut être une instance de chaîne ou d'octets ; le module lecteur `expat` produit toujours des chaînes.

> **Note**
>
> l'interface *SAX 1* précédente fournie par Python XML utilisait une interface plus proche de Java pour cette méthode. Étant donné que la plupart des analyseurs utilisés à partir de Python ne tiraient pas parti de l'ancienne interface, la signature la plus simple a été choisie pour la remplacer. Pour convertir l'ancien code vers la nouvelle interface, utilisez *content* au lieu de découper le contenu avec les anciens paramètres *offset* et *length*.

`ContentHandler.`**`ignorableWhitespace`**(*whitespace*)

Reçoit une notification en cas d'espaces pouvant être ignorées dans le contenu de l'élément.

Les analyseurs validateurs doivent utiliser cette méthode pour signaler chaque morceau d'espaces pouvant être ignorées (voir la recommandation W3C XML 1.0, section 2.10) : les analyseurs non validateurs peuvent également utiliser cette méthode s'ils sont capables d'analyser et d'utiliser des modèles de contenu.

Les analyseurs *SAX* peuvent renvoyer tous les espaces contigües en un seul morceau ou ils peuvent le diviser en plusieurs morceaux ; cependant, tous les caractères d'un événement doivent provenir de la même entité externe, afin que le localisateur fournisse des informations utiles.

`ContentHandler.`**`processingInstruction`**(*target*, *data*)

Reçoit une notification correspondant à une instruction de traitement.

L'analyseur invoque cette méthode une fois pour chaque instruction de traitement trouvée : notez que les instructions de traitement peuvent survenir avant ou après l'élément principal du document.

Un analyseur *SAX* ne doit jamais rapporter une déclaration XML (XML 1.0, section 2.8) ou une déclaration texte (XML 1.0, section 4.3.1) en utilisant cette méthode.

`ContentHandler.`**`skippedEntity`**(*name*)

Reçoit une notification correspondant à une entité ignorée.

L'analyseur invoque cette méthode une fois pour chaque entité ignorée. Les processeurs non validateurs peuvent ignorer les entités s'ils n'ont pas vu les déclarations (parce que, par exemple, l'entité a été déclarée dans un sous-ensemble *DTD* externe). Tous les processeurs peuvent ignorer les entités externes, en fonction des valeurs des propriétés `feature_external_ges` et `feature_external_pes`.

21.10.2 Objets *DTDHandler*

Les instances `DTDHandler` fournissent les méthodes suivantes :

`DTDHandler.`**`notationDecl`**(*name*, *publicId*, *systemId*)

Gère un événement de déclaration de notation.

`DTDHandler.`**`unparsedEntityDecl`**(*name*, *publicId*, *systemId*, *ndata*)

Gère un événement de déclaration d'entité non analysé.

21.10.3 Objets *EntityResolver*

`EntityResolver.`**`resolveEntity`**(*publicId*, *systemId*)

Résout l'identifiant système d'une entité et renvoie soit l'identifiant système à partir duquel lire sous forme de chaîne, soit un *InputSource* à partir duquel lire. L'implémentation par défaut renvoie *systemId*.

21.10.4 Objets *ErrorHandler*

Objects with this interface are used to receive error and warning information from the `XMLReader`. If you create an object that implements this interface, then register the object with your `XMLReader`, the parser will call the methods in your object to report all warnings and errors. There are three levels of errors available : warnings, (possibly) recoverable errors, and unrecoverable errors. All methods take a `SAXParseException` as the only parameter. Errors and warnings may be converted to an exception by raising the passed-in exception object.

`ErrorHandler.`**`error`**(*exception*)

Appelée lorsque l'analyseur rencontre une erreur récupérable. Si cette méthode ne génère pas d'exception, l'analyse peut continuer, mais l'application ne doit pas attendre d'autres informations concernant le document. Autoriser l'analyseur à continuer peut permettre de découvrir des erreurs supplémentaires dans le document d'entrée.

`ErrorHandler.`**`fatalError`**(*exception*)

Appelée lorsque l'analyseur rencontre une erreur irrécupérable ; l'analyse devrait se terminer au retour de cette méthode.

`ErrorHandler.`**`warning`**(*exception*)

Appelée lorsque l'analyseur présente des informations d'avertissement mineures à l'application. L'analyse devrait se poursuivre lorsque cette méthode termine et les informations du document continueront d'être transmises à l'application. La levée d'une exception dans cette méthode entraîne la fin de l'analyse.

21.10.5 Objets *LexicalHandler*

Gestionnaire *SAX2* facultatif pour les événements lexicaux.

Ce gestionnaire est utilisé pour obtenir des informations lexicales relatives à un document XML. Les informations lexicales incluent des informations décrivant le codage du document utilisé et les commentaires XML intégrés dans le document, ainsi que les limites des sections pour la *DTD* et pour toutes les sections *CDATA*. Les gestionnaires lexicaux sont utilisés de la même manière que les gestionnaires de contenu.

Définissez le *LexicalHandler* d'un *XMLReader* en utilisant la méthode *setProperty* avec l'identifiant de propriété `'http://xml.org/sax/properties/lexical-handler'`.

`LexicalHandler.`**`comment`**(*content*)

Signale un commentaire n'importe où dans le document (y compris la *DTD* et en dehors de l'élément de document).

`LexicalHandler.`**`startDTD`**(*name*, *public_id*, *system_id*)

Signale le début des déclarations *DTD* si le document a une *DTD* associée.

`LexicalHandler.`**`endDTD`**()

Signale la fin de la déclaration de la *DTD*.

`LexicalHandler.startCDATA()`

Signale le début d'une section marquée *CDATA*.

Le contenu de la section marquée *CDATA* est signalé via le gestionnaire de données de type caractère.

`LexicalHandler.endCDATA()`

Signale la fin d'une section marquée *CDATA*.

21.11 `xml.sax.saxutils` --- SAX Utilities

Code source : Lib/xml/sax/saxutils.py

Le module `xml.sax.saxutils` contient des classes et fonctions qui sont fréquemment utiles en créant des applications SAX, soit en utilisation directe, soit en classes de base.

`xml.sax.saxutils.escape(data, entities={})`

Échappe `'&'`, `'<'`, et `'>'` dans une chaine de caractères de données.

Vous pouvez échapper d'autres chaines de caractères de données en passant un dictionnaire au paramètre optionnel *entities*. Les clés et valeurs doivent toutes être des chaines de caractères ; chaque clé sera remplacée par sa valeur correspondante. Les caractères `'&'`, `'<'` et `'>'` sont toujours échappés même si *entities* est donné en paramètre.

> **Note**
>
> This function should only be used to escape characters that can't be used directly in XML. Do not use this function as a general string translation function.

`xml.sax.saxutils.unescape(data, entities={})`

Parse `'&'`, `'<'`, et `'>'` dans une chaine de caractères de données.

Vous pouvez dé-échapper d'autres chaines de caractères de données en passant un dictionnaire au paramètre optionnel *entities*. Les clés et valeurs doivent toutes être des chaines de caractères ; chaque clé sera remplacée par sa valeur correspondante. Les caractères `'&'`, `'<'` et `'>'` sont toujours dé-échappés même si *entities* est donné en paramètre.

`xml.sax.saxutils.quoteattr(data, entities={})`

Similaire à `escape()`, mais prépare aussi *data* pour être utilisé comme une valeur d'attribut. La valeur renvoyée est une version entre guillemets de *data* avec tous les remplacements supplémentaires nécessaires. `quoteattr()` va sélectionner un caractère guillemet basé sur le contenu de *data*, en essayant d'éviter d'encoder tous les caractères guillemets dans la chaine de caractères. Si les caractères guillemet simple et guillemets sont déjà dans *data*, les caractères guillemets simples seront encodés et *data* sera entouré de guillemets. La chaine de caractères résultante pourra être utilisée en tant que valeur d'attribut :

```
>>> print("<element attr=%s>" % quoteattr("ab ' cd \" ef"))
<element attr="ab ' cd &quot; ef">
```

Cette fonction est utile quand vous générez des valeurs d'attributs pour du HTML ou n'importe quel SGML en utilisant la syntaxe concrète de référence.

class `xml.sax.saxutils.XMLGenerator(out=None, encoding='iso-8859-1', short_empty_elements=False)`

Cette classe implémente l'interface `ContentHandler` en écrivant les événements SAX dans un document XML. En d'autres termes, utiliser un `XMLGenerator` en tant que gestionnaire de contenu reproduira le document original qui était analysé. *out* devrait être un objet de type fichier qui est par défaut *sys.stdout*. *encoding* est l'encodage du flux de sortie qui est par défaut `'iso-8859-1'`. *short_empty_elements* contrôle le formatage des éléments qui ne contiennent rien : si `False` (par défaut), ils sont émis comme une paire de balises (début, fin). Si la valeur est `True`, ils sont émis comme une balise seule auto-fermante.

Modifié dans la version 3.2 : Added the *short_empty_elements* parameter.

class xml.sax.saxutils.**XMLFilterBase**(*base*)

Cette classe est faite pour être entre `XMLReader` et le gestionnaire des événements de l'application client. Par défaut, elle ne fait rien mais passe les requêtes au lecteur et les événements au gestionnaire sans les modifier, mais des sous-classes peuvent surcharger des méthodes spécifiques pour modifier le flux d'événements ou la configuration des requêtes à leur passage.

xml.sax.saxutils.**prepare_input_source**(*source*, *base=''*)

This function takes an input source and an optional base URL and returns a fully resolved `InputSource` object ready for reading. The input source can be given as a string, a file-like object, or an `InputSource` object ; parsers will use this function to implement the polymorphic *source* argument to their `parse()` method.

21.12 xml.sax.xmlreader --- Interface for XML parsers

Source code : Lib/xml/sax/xmlreader.py

SAX parsers implement the `XMLReader` interface. They are implemented in a Python module, which must provide a function `create_parser()`. This function is invoked by `xml.sax.make_parser()` with no arguments to create a new parser object.

class xml.sax.xmlreader.**XMLReader**

Base class which can be inherited by SAX parsers.

class xml.sax.xmlreader.**IncrementalParser**

In some cases, it is desirable not to parse an input source at once, but to feed chunks of the document as they get available. Note that the reader will normally not read the entire file, but read it in chunks as well ; still `parse()` won't return until the entire document is processed. So these interfaces should be used if the blocking behaviour of `parse()` is not desirable.

When the parser is instantiated it is ready to begin accepting data from the feed method immediately. After parsing has been finished with a call to close the reset method must be called to make the parser ready to accept new data, either from feed or using the parse method.

Note that these methods must *not* be called during parsing, that is, after parse has been called and before it returns.

By default, the class also implements the parse method of the XMLReader interface using the feed, close and reset methods of the IncrementalParser interface as a convenience to SAX 2.0 driver writers.

class xml.sax.xmlreader.**Locator**

Interface for associating a SAX event with a document location. A locator object will return valid results only during calls to DocumentHandler methods ; at any other time, the results are unpredictable. If information is not available, methods may return `None`.

class xml.sax.xmlreader.**InputSource**(*system_id=None*)

Encapsulation of the information needed by the `XMLReader` to read entities.

This class may include information about the public identifier, system identifier, byte stream (possibly with character encoding information) and/or the character stream of an entity.

Applications will create objects of this class for use in the `XMLReader.parse()` method and for returning from EntityResolver.resolveEntity.

An `InputSource` belongs to the application, the `XMLReader` is not allowed to modify `InputSource` objects passed to it from the application, although it may make copies and modify those.

class xml.sax.xmlreader.**AttributesImpl**(*attrs*)

This is an implementation of the `Attributes` interface (see section *The Attributes Interface*). This is a dictionary-like object which represents the element attributes in a `startElement()` call. In addition to the most useful dictionary operations, it supports a number of other methods as described by the interface. Objects of this class should be instantiated by readers ; *attrs* must be a dictionary-like object containing a mapping from attribute names to attribute values.

class xml.sax.xmlreader.**AttributesNSImpl**(*attrs*, *qnames*)

Namespace-aware variant of `AttributesImpl`, which will be passed to `startElementNS()`. It is derived from `AttributesImpl`, but understands attribute names as two-tuples of *namespaceURI* and *localname*. In addition, it provides a number of methods expecting qualified names as they appear in the original document. This class implements the `AttributesNS` interface (see section *The AttributesNS Interface*).

21.12.1 XMLReader Objects

The `XMLReader` interface supports the following methods :

XMLReader.**parse**(*source*)

Process an input source, producing SAX events. The *source* object can be a system identifier (a string identifying the input source -- typically a file name or a URL), a `pathlib.Path` or *path-like* object, or an `InputSource` object. When `parse()` returns, the input is completely processed, and the parser object can be discarded or reset.

Modifié dans la version 3.5 : Added support of character streams.

Modifié dans la version 3.8 : Added support of path-like objects.

XMLReader.**getContentHandler**()

Return the current `ContentHandler`.

XMLReader.**setContentHandler**(*handler*)

Set the current `ContentHandler`. If no `ContentHandler` is set, content events will be discarded.

XMLReader.**getDTDHandler**()

Return the current `DTDHandler`.

XMLReader.**setDTDHandler**(*handler*)

Set the current `DTDHandler`. If no `DTDHandler` is set, DTD events will be discarded.

XMLReader.**getEntityResolver**()

Return the current `EntityResolver`.

XMLReader.**setEntityResolver**(*handler*)

Set the current `EntityResolver`. If no `EntityResolver` is set, attempts to resolve an external entity will result in opening the system identifier for the entity, and fail if it is not available.

XMLReader.**getErrorHandler**()

Return the current `ErrorHandler`.

XMLReader.**setErrorHandler**(*handler*)

Set the current error handler. If no `ErrorHandler` is set, errors will be raised as exceptions, and warnings will be printed.

XMLReader.**setLocale**(*locale*)

Allow an application to set the locale for errors and warnings.

SAX parsers are not required to provide localization for errors and warnings; if they cannot support the requested locale, however, they must raise a SAX exception. Applications may request a locale change in the middle of a parse.

XMLReader.**getFeature**(*featurename*)

Return the current setting for feature *featurename*. If the feature is not recognized, `SAXNotRecognizedException` is raised. The well-known featurenames are listed in the module `xml.sax.handler`.

XMLReader.**setFeature**(*featurename*, *value*)

Set the *featurename* to *value*. If the feature is not recognized, `SAXNotRecognizedException` is raised. If the feature or its setting is not supported by the parser, *SAXNotSupportedException* is raised.

`XMLReader.getProperty(propertyname)`

Return the current setting for property *propertyname*. If the property is not recognized, a `SAXNotRecognizedException` is raised. The well-known propertynames are listed in the module `xml.sax.handler`.

`XMLReader.setProperty(propertyname, value)`

Set the *propertyname* to *value*. If the property is not recognized, `SAXNotRecognizedException` is raised. If the property or its setting is not supported by the parser, *SAXNotSupportedException* is raised.

21.12.2 IncrementalParser Objects

Instances of `IncrementalParser` offer the following additional methods :

`IncrementalParser.feed(data)`

Process a chunk of *data*.

`IncrementalParser.close()`

Assume the end of the document. That will check well-formedness conditions that can be checked only at the end, invoke handlers, and may clean up resources allocated during parsing.

`IncrementalParser.reset()`

This method is called after close has been called to reset the parser so that it is ready to parse new documents. The results of calling parse or feed after close without calling reset are undefined.

21.12.3 Locator Objects

Instances of `Locator` provide these methods :

`Locator.getColumnNumber()`

Return the column number where the current event begins.

`Locator.getLineNumber()`

Return the line number where the current event begins.

`Locator.getPublicId()`

Return the public identifier for the current event.

`Locator.getSystemId()`

Return the system identifier for the current event.

21.12.4 InputSource Objects

`InputSource.setPublicId(id)`

Sets the public identifier of this `InputSource`.

`InputSource.getPublicId()`

Returns the public identifier of this `InputSource`.

`InputSource.setSystemId(id)`

Sets the system identifier of this `InputSource`.

`InputSource.getSystemId()`

Returns the system identifier of this `InputSource`.

`InputSource.setEncoding(encoding)`

Sets the character encoding of this `InputSource`.

The encoding must be a string acceptable for an XML encoding declaration (see section 4.3.3 of the XML recommendation).

The encoding attribute of the `InputSource` is ignored if the `InputSource` also contains a character stream.

`InputSource.getEncoding()`

Get the character encoding of this InputSource.

`InputSource.setByteStream(bytefile)`

Set the byte stream (a *binary file*) for this input source.

The SAX parser will ignore this if there is also a character stream specified, but it will use a byte stream in preference to opening a URI connection itself.

If the application knows the character encoding of the byte stream, it should set it with the setEncoding method.

`InputSource.getByteStream()`

Get the byte stream for this input source.

The getEncoding method will return the character encoding for this byte stream, or `None` if unknown.

`InputSource.setCharacterStream(charfile)`

Set the character stream (a *text file*) for this input source.

If there is a character stream specified, the SAX parser will ignore any byte stream and will not attempt to open a URI connection to the system identifier.

`InputSource.getCharacterStream()`

Get the character stream for this input source.

21.12.5 The `Attributes` Interface

`Attributes` objects implement a portion of the *mapping protocol*, including the methods `copy()`, `get()`, `__contains__()`, `items()`, `keys()`, and `values()`. The following methods are also provided :

`Attributes.getLength()`

Return the number of attributes.

`Attributes.getNames()`

Return the names of the attributes.

`Attributes.getType(name)`

Returns the type of the attribute *name*, which is normally `'CDATA'`.

`Attributes.getValue(name)`

Return the value of attribute *name*.

21.12.6 The `AttributesNS` Interface

This interface is a subtype of the `Attributes` interface (see section *The Attributes Interface*). All methods supported by that interface are also available on `AttributesNS` objects.

The following methods are also available :

`AttributesNS.getValueByQName(name)`

Return the value for a qualified name.

`AttributesNS.getNameByQName(name)`

Return the `(namespace, localname)` pair for a qualified *name*.

`AttributesNS.getQNameByName(name)`

Return the qualified name for a `(namespace, localname)` pair.

`AttributesNS.getQNames()`

Return the qualified names of all attributes.

21.13 xml.parsers.expat --- Fast XML parsing using Expat

Avertissement

The pyexpat module is not secure against maliciously constructed data. If you need to parse untrusted or unauthenticated data see *Vulnérabilités XML*.

The *xml.parsers.expat* module is a Python interface to the Expat non-validating XML parser. The module provides a single extension type, xmlparser, that represents the current state of an XML parser. After an xmlparser object has been created, various attributes of the object can be set to handler functions. When an XML document is then fed to the parser, the handler functions are called for the character data and markup in the XML document.

This module uses the pyexpat module to provide access to the Expat parser. Direct use of the pyexpat module is deprecated.

This module provides one exception and one type object :

exception xml.parsers.expat.**ExpatError**

The exception raised when Expat reports an error. See section *ExpatError Exceptions* for more information on interpreting Expat errors.

exception xml.parsers.expat.**error**

Alias for *ExpatError*.

xml.parsers.expat.**XMLParserType**

The type of the return values from the *ParserCreate()* function.

The *xml.parsers.expat* module contains two functions :

xml.parsers.expat.**ErrorString**(*errno*)

Returns an explanatory string for a given error number *errno*.

xml.parsers.expat.**ParserCreate**(*encoding=None*, *namespace_separator=None*)

Creates and returns a new xmlparser object. *encoding*, if specified, must be a string naming the encoding used by the XML data. Expat doesn't support as many encodings as Python does, and its repertoire of encodings can't be extended ; it supports UTF-8, UTF-16, ISO-8859-1 (Latin1), and ASCII. If *encoding* [1] is given it will override the implicit or explicit encoding of the document.

Expat can optionally do XML namespace processing for you, enabled by providing a value for *namespace_separator*. The value must be a one-character string ; a *ValueError* will be raised if the string has an illegal length (None is considered the same as omission). When namespace processing is enabled, element type names and attribute names that belong to a namespace will be expanded. The element name passed to the element handlers StartElementHandler and EndElementHandler will be the concatenation of the namespace URI, the namespace separator character, and the local part of the name. If the namespace separator is a zero byte (chr(0)) then the namespace URI and the local part will be concatenated without any separator.

For example, if *namespace_separator* is set to a space character (' ') and the following document is parsed :

```
<?xml version="1.0"?>
<root xmlns    = "http://default-namespace.org/"
      xmlns:py = "http://www.python.org/ns/">
  <py:elem1 />
  <elem2 xmlns="" />
</root>
```

StartElementHandler will receive the following strings for each element :

1. The encoding string included in XML output should conform to the appropriate standards. For example, "UTF-8" is valid, but "UTF8" is not. See https://www.w3.org/TR/2006/REC-xml11-20060816/#NT-EncodingDecl and https://www.iana.org/assignments/character-sets/character-sets.xhtml.

```
http://default-namespace.org/ root
http://www.python.org/ns/ elem1
elem2
```

Due to limitations in the `Expat` library used by `pyexpat`, the `xmlparser` instance returned can only be used to parse a single XML document. Call `ParserCreate` for each document to provide unique parser instances.

Voir aussi

The Expat XML Parser
Home page of the Expat project.

21.13.1 Objets *XMLParser*

`xmlparser` objects have the following methods :

`xmlparser.`**`Parse`**(*data*[, *isfinal*])

Parses the contents of the string *data*, calling the appropriate handler functions to process the parsed data. *isfinal* must be true on the final call to this method ; it allows the parsing of a single file in fragments, not the submission of multiple files. *data* can be the empty string at any time.

`xmlparser.`**`ParseFile`**(*file*)

Parse XML data reading from the object *file*. *file* only needs to provide the `read(nbytes)` method, returning the empty string when there's no more data.

`xmlparser.`**`SetBase`**(*base*)

Sets the base to be used for resolving relative URIs in system identifiers in declarations. Resolving relative identifiers is left to the application : this value will be passed through as the *base* argument to the `ExternalEntityRefHandler()`, `NotationDeclHandler()`, and `UnparsedEntityDeclHandler()` functions.

`xmlparser.`**`GetBase`**()

Returns a string containing the base set by a previous call to `SetBase()`, or `None` if `SetBase()` hasn't been called.

`xmlparser.`**`GetInputContext`**()

Returns the input data that generated the current event as a string. The data is in the encoding of the entity which contains the text. When called while an event handler is not active, the return value is `None`.

`xmlparser.`**`ExternalEntityParserCreate`**(*context*[, *encoding*])

Create a "child" parser which can be used to parse an external parsed entity referred to by content parsed by the parent parser. The *context* parameter should be the string passed to the `ExternalEntityRefHandler()` handler function, described below. The child parser is created with the `ordered_attributes` and `specified_attributes` set to the values of this parser.

`xmlparser.`**`SetParamEntityParsing`**(*flag*)

Control parsing of parameter entities (including the external DTD subset). Possible *flag* values are `XML_PARAM_ENTITY_PARSING_NEVER`, `XML_PARAM_ENTITY_PARSING_UNLESS_STANDALONE` and `XML_PARAM_ENTITY_PARSING_ALWAYS`. Return true if setting the flag was successful.

`xmlparser.`**`UseForeignDTD`**([*flag*])

Calling this with a true value for *flag* (the default) will cause Expat to call the `ExternalEntityRefHandler` with `None` for all arguments to allow an alternate DTD to be loaded. If the document does not contain a document type declaration, the `ExternalEntityRefHandler` will still be called, but the `StartDoctypeDeclHandler` and `EndDoctypeDeclHandler` will not be called.

Passing a false value for *flag* will cancel a previous call that passed a true value, but otherwise has no effect.

This method can only be called before the `Parse()` or `ParseFile()` methods are called ; calling it after either of those have been called causes `ExpatError` to be raised with the `code` attribute set to `errors.codes[errors.XML_ERROR_CANT_CHANGE_FEATURE_ONCE_PARSING]`.

`xmlparser.`**`SetReparseDeferralEnabled`**(*enabled*)

> **Avertissement**
>
> Calling `SetReparseDeferralEnabled(False)` has security implications, as detailed below ; please make sure to understand these consequences prior to using the `SetReparseDeferralEnabled` method.

Expat 2.6.0 introduced a security mechanism called "reparse deferral" where instead of causing denial of service through quadratic runtime from reparsing large tokens, reparsing of unfinished tokens is now delayed by default until a sufficient amount of input is reached. Due to this delay, registered handlers may — depending of the sizing of input chunks pushed to Expat — no longer be called right after pushing new input to the parser. Where immediate feedback and taking over responsibility of protecting against denial of service from large tokens are both wanted, calling `SetReparseDeferralEnabled(False)` disables reparse deferral for the current Expat parser instance, temporarily or altogether. Calling `SetReparseDeferralEnabled(True)` allows re-enabling reparse deferral.

Note that `SetReparseDeferralEnabled()` has been backported to some prior releases of CPython as a security fix. Check for availability of `SetReparseDeferralEnabled()` using `hasattr()` if used in code running across a variety of Python versions.

Ajouté dans la version 3.13.

`xmlparser.`**`GetReparseDeferralEnabled`**()

Returns whether reparse deferral is currently enabled for the given Expat parser instance.

Ajouté dans la version 3.13.

`xmlparser` objects have the following attributes :

`xmlparser.`**`buffer_size`**

The size of the buffer used when `buffer_text` is true. A new buffer size can be set by assigning a new integer value to this attribute. When the size is changed, the buffer will be flushed.

`xmlparser.`**`buffer_text`**

Setting this to true causes the `xmlparser` object to buffer textual content returned by Expat to avoid multiple calls to the `CharacterDataHandler()` callback whenever possible. This can improve performance substantially since Expat normally breaks character data into chunks at every line ending. This attribute is false by default, and may be changed at any time. Note that when it is false, data that does not contain newlines may be chunked too.

`xmlparser.`**`buffer_used`**

If `buffer_text` is enabled, the number of bytes stored in the buffer. These bytes represent UTF-8 encoded text. This attribute has no meaningful interpretation when `buffer_text` is false.

`xmlparser.`**`ordered_attributes`**

Setting this attribute to a non-zero integer causes the attributes to be reported as a list rather than a dictionary. The attributes are presented in the order found in the document text. For each attribute, two list entries are presented : the attribute name and the attribute value. (Older versions of this module also used this format.) By default, this attribute is false ; it may be changed at any time.

`xmlparser.`**`specified_attributes`**

If set to a non-zero integer, the parser will report only those attributes which were specified in the document instance and not those which were derived from attribute declarations. Applications which set this need to be especially careful to use what additional information is available from the declarations as needed to comply with the standards for the behavior of XML processors. By default, this attribute is false ; it may be changed at any time.

The following attributes contain values relating to the most recent error encountered by an `xmlparser` object, and will only have correct values once a call to `Parse()` or `ParseFile()` has raised an *xml.parsers.expat.ExpatError* exception.

`xmlparser.`**`ErrorByteIndex`**

Byte index at which an error occurred.

`xmlparser.`**`ErrorCode`**

Numeric code specifying the problem. This value can be passed to the *ErrorString()* function, or compared to one of the constants defined in the `errors` object.

`xmlparser.`**`ErrorColumnNumber`**

Column number at which an error occurred.

`xmlparser.`**`ErrorLineNumber`**

Line number at which an error occurred.

The following attributes contain values relating to the current parse location in an `xmlparser` object. During a callback reporting a parse event they indicate the location of the first of the sequence of characters that generated the event. When called outside of a callback, the position indicated will be just past the last parse event (regardless of whether there was an associated callback).

`xmlparser.`**`CurrentByteIndex`**

Current byte index in the parser input.

`xmlparser.`**`CurrentColumnNumber`**

Current column number in the parser input.

`xmlparser.`**`CurrentLineNumber`**

Current line number in the parser input.

Here is the list of handlers that can be set. To set a handler on an `xmlparser` object *o*, use `o.handlername = func`. *handlername* must be taken from the following list, and *func* must be a callable object accepting the correct number of arguments. The arguments are all strings, unless otherwise stated.

`xmlparser.`**`XmlDeclHandler`**(*version*, *encoding*, *standalone*)

Called when the XML declaration is parsed. The XML declaration is the (optional) declaration of the applicable version of the XML recommendation, the encoding of the document text, and an optional "standalone" declaration. *version* and *encoding* will be strings, and *standalone* will be `1` if the document is declared standalone, `0` if it is declared not to be standalone, or `-1` if the standalone clause was omitted. This is only available with Expat version 1.95.0 or newer.

`xmlparser.`**`StartDoctypeDeclHandler`**(*doctypeName*, *systemId*, *publicId*, *has_internal_subset*)

Called when Expat begins parsing the document type declaration (`<!DOCTYPE ...`). The *doctypeName* is provided exactly as presented. The *systemId* and *publicId* parameters give the system and public identifiers if specified, or `None` if omitted. *has_internal_subset* will be true if the document contains and internal document declaration subset. This requires Expat version 1.2 or newer.

`xmlparser.`**`EndDoctypeDeclHandler`**()

Called when Expat is done parsing the document type declaration. This requires Expat version 1.2 or newer.

`xmlparser.`**`ElementDeclHandler`**(*name*, *model*)

Called once for each element type declaration. *name* is the name of the element type, and *model* is a representation of the content model.

`xmlparser.`**`AttlistDeclHandler`**(*elname*, *attname*, *type*, *default*, *required*)

Called for each declared attribute for an element type. If an attribute list declaration declares three attributes, this handler is called three times, once for each attribute. *elname* is the name of the element to which the declaration applies and *attname* is the name of the attribute declared. The attribute type is a string passed as *type*; the possible values are `'CDATA'`, `'ID'`, `'IDREF'`, ... *default* gives the default value for the attribute used when the attribute is not specified by the document instance, or `None` if there is no default value (`#IMPLIED` values). If the attribute is required to be given in the document instance, *required* will be true. This requires Expat version 1.95.0 or newer.

`xmlparser.StartElementHandler(name, attributes)`

Called for the start of every element. *name* is a string containing the element name, and *attributes* is the element attributes. If `ordered_attributes` is true, this is a list (see `ordered_attributes` for a full description). Otherwise it's a dictionary mapping names to values.

`xmlparser.EndElementHandler(name)`

Called for the end of every element.

`xmlparser.ProcessingInstructionHandler(target, data)`

Called for every processing instruction.

`xmlparser.CharacterDataHandler(data)`

Called for character data. This will be called for normal character data, CDATA marked content, and ignorable whitespace. Applications which must distinguish these cases can use the `StartCdataSectionHandler`, `EndCdataSectionHandler`, and `ElementDeclHandler` callbacks to collect the required information. Note that the character data may be chunked even if it is short and so you may receive more than one call to `CharacterDataHandler()`. Set the `buffer_text` instance attribute to `True` to avoid that.

`xmlparser.UnparsedEntityDeclHandler(entityName, base, systemId, publicId, notationName)`

Called for unparsed (NDATA) entity declarations. This is only present for version 1.2 of the Expat library; for more recent versions, use `EntityDeclHandler` instead. (The underlying function in the Expat library has been declared obsolete.)

`xmlparser.EntityDeclHandler(entityName, is_parameter_entity, value, base, systemId, publicId, notationName)`

Called for all entity declarations. For parameter and internal entities, *value* will be a string giving the declared contents of the entity; this will be `None` for external entities. The *notationName* parameter will be `None` for parsed entities, and the name of the notation for unparsed entities. *is_parameter_entity* will be true if the entity is a parameter entity or false for general entities (most applications only need to be concerned with general entities). This is only available starting with version 1.95.0 of the Expat library.

`xmlparser.NotationDeclHandler(notationName, base, systemId, publicId)`

Called for notation declarations. *notationName*, *base*, and *systemId*, and *publicId* are strings if given. If the public identifier is omitted, *publicId* will be `None`.

`xmlparser.StartNamespaceDeclHandler(prefix, uri)`

Called when an element contains a namespace declaration. Namespace declarations are processed before the `StartElementHandler` is called for the element on which declarations are placed.

`xmlparser.EndNamespaceDeclHandler(prefix)`

Called when the closing tag is reached for an element that contained a namespace declaration. This is called once for each namespace declaration on the element in the reverse of the order for which the `StartNamespaceDeclHandler` was called to indicate the start of each namespace declaration's scope. Calls to this handler are made after the corresponding `EndElementHandler` for the end of the element.

`xmlparser.CommentHandler(data)`

Called for comments. *data* is the text of the comment, excluding the leading `'<!--'` and trailing `'-->'`.

`xmlparser.StartCdataSectionHandler()`

Called at the start of a CDATA section. This and `EndCdataSectionHandler` are needed to be able to identify the syntactical start and end for CDATA sections.

`xmlparser.EndCdataSectionHandler()`

Called at the end of a CDATA section.

`xmlparser.DefaultHandler(data)`

Called for any characters in the XML document for which no applicable handler has been specified. This means characters that are part of a construct which could be reported, but for which no handler has been supplied.

xmlparser.**DefaultHandlerExpand**(*data*)

This is the same as the `DefaultHandler()`, but doesn't inhibit expansion of internal entities. The entity reference will not be passed to the default handler.

xmlparser.**NotStandaloneHandler**()

Called if the XML document hasn't been declared as being a standalone document. This happens when there is an external subset or a reference to a parameter entity, but the XML declaration does not set standalone to `yes` in an XML declaration. If this handler returns `0`, then the parser will raise an `XML_ERROR_NOT_STANDALONE` error. If this handler is not set, no exception is raised by the parser for this condition.

xmlparser.**ExternalEntityRefHandler**(*context*, *base*, *systemId*, *publicId*)

Called for references to external entities. *base* is the current base, as set by a previous call to `SetBase()`. The public and system identifiers, *systemId* and *publicId*, are strings if given ; if the public identifier is not given, *publicId* will be `None`. The *context* value is opaque and should only be used as described below.

For external entities to be parsed, this handler must be implemented. It is responsible for creating the sub-parser using `ExternalEntityParserCreate(context)`, initializing it with the appropriate callbacks, and parsing the entity. This handler should return an integer ; if it returns `0`, the parser will raise an `XML_ERROR_EXTERNAL_ENTITY_HANDLING` error, otherwise parsing will continue.

If this handler is not provided, external entities are reported by the `DefaultHandler` callback, if provided.

21.13.2 ExpatError Exceptions

`ExpatError` exceptions have a number of interesting attributes :

ExpatError.**code**

Expat's internal error number for the specific error. The `errors.messages` dictionary maps these error numbers to Expat's error messages. For example :

```
from xml.parsers.expat import ParserCreate, ExpatError, errors

p = ParserCreate()
try:
    p.Parse(some_xml_document)
except ExpatError as err:
    print("Error:", errors.messages[err.code])
```

The `errors` module also provides error message constants and a dictionary `codes` mapping these messages back to the error codes, see below.

ExpatError.**lineno**

Line number on which the error was detected. The first line is numbered `1`.

ExpatError.**offset**

Character offset into the line where the error occurred. The first column is numbered `0`.

21.13.3 Exemple

The following program defines three handlers that just print out their arguments.

```
import xml.parsers.expat

# 3 handler functions
def start_element(name, attrs):
    print('Start element:', name, attrs)
def end_element(name):
    print('End element:', name)
def char_data(data):
    print('Character data:', repr(data))
```

(suite sur la page suivante)

(suite de la page précédente)

```
p = xml.parsers.expat.ParserCreate()

p.StartElementHandler = start_element
p.EndElementHandler = end_element
p.CharacterDataHandler = char_data

p.Parse("""<?xml version="1.0"?>
<parent id="top"><child1 name="paul">Text goes here</child1>
<child2 name="fred">More text</child2>
</parent>""", 1)
```

The output from this program is :

```
Start element: parent {'id': 'top'}
Start element: child1 {'name': 'paul'}
Character data: 'Text goes here'
End element: child1
Character data: '\n'
Start element: child2 {'name': 'fred'}
Character data: 'More text'
End element: child2
Character data: '\n'
End element: parent
```

21.13.4 Content Model Descriptions

Content models are described using nested tuples. Each tuple contains four values : the type, the quantifier, the name, and a tuple of children. Children are simply additional content model descriptions.

The values of the first two fields are constants defined in the *xml.parsers.expat.model* module. These constants can be collected in two groups : the model type group and the quantifier group.

The constants in the model type group are :

xml.parsers.expat.model.**XML_CTYPE_ANY**

The element named by the model name was declared to have a content model of `ANY`.

xml.parsers.expat.model.**XML_CTYPE_CHOICE**

The named element allows a choice from a number of options ; this is used for content models such as `(A | B | C)`.

xml.parsers.expat.model.**XML_CTYPE_EMPTY**

Elements which are declared to be `EMPTY` have this model type.

xml.parsers.expat.model.**XML_CTYPE_MIXED**

xml.parsers.expat.model.**XML_CTYPE_NAME**

xml.parsers.expat.model.**XML_CTYPE_SEQ**

Models which represent a series of models which follow one after the other are indicated with this model type. This is used for models such as `(A, B, C)`.

The constants in the quantifier group are :

xml.parsers.expat.model.**XML_CQUANT_NONE**

No modifier is given, so it can appear exactly once, as for `A`.

xml.parsers.expat.model.**XML_CQUANT_OPT**

The model is optional : it can appear once or not at all, as for `A?`.

xml.parsers.expat.model.**XML_CQUANT_PLUS**

The model must occur one or more times (like `A+`).

xml.parsers.expat.model.**XML_CQUANT_REP**

The model must occur zero or more times, as for `A*`.

21.13.5 Expat error constants

The following constants are provided in the *xml.parsers.expat.errors* module. These constants are useful in interpreting some of the attributes of the `ExpatError` exception objects raised when an error has occurred. Since for backwards compatibility reasons, the constants' value is the error *message* and not the numeric error *code*, you do this by comparing its *code* attribute with `errors.codes[errors.XML_ERROR_`*CONSTANT_NAME*`]`.

The `errors` module has the following attributes :

xml.parsers.expat.errors.**codes**

A dictionary mapping string descriptions to their error codes.

Ajouté dans la version 3.2.

xml.parsers.expat.errors.**messages**

A dictionary mapping numeric error codes to their string descriptions.

Ajouté dans la version 3.2.

xml.parsers.expat.errors.**XML_ERROR_ASYNC_ENTITY**

xml.parsers.expat.errors.**XML_ERROR_ATTRIBUTE_EXTERNAL_ENTITY_REF**

An entity reference in an attribute value referred to an external entity instead of an internal entity.

xml.parsers.expat.errors.**XML_ERROR_BAD_CHAR_REF**

A character reference referred to a character which is illegal in XML (for example, character `0`, or '`&#0;`').

xml.parsers.expat.errors.**XML_ERROR_BINARY_ENTITY_REF**

An entity reference referred to an entity which was declared with a notation, so cannot be parsed.

xml.parsers.expat.errors.**XML_ERROR_DUPLICATE_ATTRIBUTE**

An attribute was used more than once in a start tag.

xml.parsers.expat.errors.**XML_ERROR_INCORRECT_ENCODING**

xml.parsers.expat.errors.**XML_ERROR_INVALID_TOKEN**

Raised when an input byte could not properly be assigned to a character ; for example, a NUL byte (value `0`) in a UTF-8 input stream.

xml.parsers.expat.errors.**XML_ERROR_JUNK_AFTER_DOC_ELEMENT**

Something other than whitespace occurred after the document element.

xml.parsers.expat.errors.**XML_ERROR_MISPLACED_XML_PI**

An XML declaration was found somewhere other than the start of the input data.

xml.parsers.expat.errors.**XML_ERROR_NO_ELEMENTS**

The document contains no elements (XML requires all documents to contain exactly one top-level element)..

xml.parsers.expat.errors.**XML_ERROR_NO_MEMORY**

Expat was not able to allocate memory internally.

xml.parsers.expat.errors.**XML_ERROR_PARAM_ENTITY_REF**

A parameter entity reference was found where it was not allowed.

xml.parsers.expat.errors.**XML_ERROR_PARTIAL_CHAR**

An incomplete character was found in the input.

xml.parsers.expat.errors.**XML_ERROR_RECURSIVE_ENTITY_REF**

An entity reference contained another reference to the same entity; possibly via a different name, and possibly indirectly.

xml.parsers.expat.errors.**XML_ERROR_SYNTAX**

Some unspecified syntax error was encountered.

xml.parsers.expat.errors.**XML_ERROR_TAG_MISMATCH**

An end tag did not match the innermost open start tag.

xml.parsers.expat.errors.**XML_ERROR_UNCLOSED_TOKEN**

Some token (such as a start tag) was not closed before the end of the stream or the next token was encountered.

xml.parsers.expat.errors.**XML_ERROR_UNDEFINED_ENTITY**

A reference was made to an entity which was not defined.

xml.parsers.expat.errors.**XML_ERROR_UNKNOWN_ENCODING**

The document encoding is not supported by Expat.

xml.parsers.expat.errors.**XML_ERROR_UNCLOSED_CDATA_SECTION**

A CDATA marked section was not closed.

xml.parsers.expat.errors.**XML_ERROR_EXTERNAL_ENTITY_HANDLING**

xml.parsers.expat.errors.**XML_ERROR_NOT_STANDALONE**

The parser determined that the document was not "standalone" though it declared itself to be in the XML declaration, and the `NotStandaloneHandler` was set and returned `0`.

xml.parsers.expat.errors.**XML_ERROR_UNEXPECTED_STATE**

xml.parsers.expat.errors.**XML_ERROR_ENTITY_DECLARED_IN_PE**

xml.parsers.expat.errors.**XML_ERROR_FEATURE_REQUIRES_XML_DTD**

An operation was requested that requires DTD support to be compiled in, but Expat was configured without DTD support. This should never be reported by a standard build of the *xml.parsers.expat* module.

xml.parsers.expat.errors.**XML_ERROR_CANT_CHANGE_FEATURE_ONCE_PARSING**

A behavioral change was requested after parsing started that can only be changed before parsing has started. This is (currently) only raised by `UseForeignDTD()`.

xml.parsers.expat.errors.**XML_ERROR_UNBOUND_PREFIX**

An undeclared prefix was found when namespace processing was enabled.

xml.parsers.expat.errors.**XML_ERROR_UNDECLARING_PREFIX**

The document attempted to remove the namespace declaration associated with a prefix.

xml.parsers.expat.errors.**XML_ERROR_INCOMPLETE_PE**

A parameter entity contained incomplete markup.

xml.parsers.expat.errors.**XML_ERROR_XML_DECL**

The document contained no document element at all.

xml.parsers.expat.errors.**XML_ERROR_TEXT_DECL**

There was an error parsing a text declaration in an external entity.

xml.parsers.expat.errors.**XML_ERROR_PUBLICID**

Characters were found in the public id that are not allowed.

xml.parsers.expat.errors.**XML_ERROR_SUSPENDED**

The requested operation was made on a suspended parser, but isn't allowed. This includes attempts to provide additional input or to stop the parser.

`xml.parsers.expat.errors.`**`XML_ERROR_NOT_SUSPENDED`**

An attempt to resume the parser was made when the parser had not been suspended.

`xml.parsers.expat.errors.`**`XML_ERROR_ABORTED`**

This should not be reported to Python applications.

`xml.parsers.expat.errors.`**`XML_ERROR_FINISHED`**

The requested operation was made on a parser which was finished parsing input, but isn't allowed. This includes attempts to provide additional input or to stop the parser.

`xml.parsers.expat.errors.`**`XML_ERROR_SUSPEND_PE`**

`xml.parsers.expat.errors.`**`XML_ERROR_RESERVED_PREFIX_XML`**

An attempt was made to undeclare reserved namespace prefix `xml` or to bind it to another namespace URI.

`xml.parsers.expat.errors.`**`XML_ERROR_RESERVED_PREFIX_XMLNS`**

An attempt was made to declare or undeclare reserved namespace prefix `xmlns`.

`xml.parsers.expat.errors.`**`XML_ERROR_RESERVED_NAMESPACE_URI`**

An attempt was made to bind the URI of one the reserved namespace prefixes `xml` and `xmlns` to another namespace prefix.

`xml.parsers.expat.errors.`**`XML_ERROR_INVALID_ARGUMENT`**

This should not be reported to Python applications.

`xml.parsers.expat.errors.`**`XML_ERROR_NO_BUFFER`**

This should not be reported to Python applications.

`xml.parsers.expat.errors.`**`XML_ERROR_AMPLIFICATION_LIMIT_BREACH`**

The limit on input amplification factor (from DTD and entities) has been breached.

`xml.parsers.expat.errors.`**`XML_ERROR_NOT_STARTED`**

The parser was tried to be stopped or suspended before it started.

Ajouté dans la version 3.14.

Notes

CHAPITRE 22

Gestion des protocoles internet

Les modules documentés dans ce chapitre implémentent des protocoles relatifs à Internet et à ses technologies relatives. Ils sont tous implémentés en Python. La majorité de ces modules nécessitent la présence du module `socket` lui même dépendant du système, mais fourni sur la plupart des plateformes populaires. Voici une vue d'ensemble :

22.1 `webbrowser` --- Convenient web-browser controller

Source code : Lib/webbrowser.py

The `webbrowser` module provides a high-level interface to allow displaying web-based documents to users. Under most circumstances, simply calling the `open()` function from this module will do the right thing.

Under Unix, graphical browsers are preferred under X11, but text-mode browsers will be used if graphical browsers are not available or an X11 display isn't available. If text-mode browsers are used, the calling process will block until the user exits the browser.

If the environment variable `BROWSER` exists, it is interpreted as the `os.pathsep`-separated list of browsers to try ahead of the platform defaults. When the value of a list part contains the string `%s`, then it is interpreted as a literal browser command line to be used with the argument URL substituted for `%s`; if the part does not contain `%s`, it is simply interpreted as the name of the browser to launch. [1]

For non-Unix platforms, or when a remote browser is available on Unix, the controlling process will not wait for the user to finish with the browser, but allow the remote browser to maintain its own windows on the display. If remote browsers are not available on Unix, the controlling process will launch a new browser and wait.

On iOS, the `BROWSER` environment variable, as well as any arguments controlling autoraise, browser preference, and new tab/window creation will be ignored. Web pages will *always* be opened in the user's preferred browser, in a new tab, with the browser being brought to the foreground. The use of the `webbrowser` module on iOS requires the `ctypes` module. If `ctypes` isn't available, calls to `open()` will fail.

The script **`webbrowser`** can be used as a command-line interface for the module. It accepts a URL as the argument. It accepts the following optional parameters :

— `-n`/`--new-window` opens the URL in a new browser window, if possible.
— `-t`/`--new-tab` opens the URL in a new browser page ("tab").

The options are, naturally, mutually exclusive. Usage example :

1. Executables named here without a full path will be searched in the directories given in the `PATH` environment variable.

```
python -m webbrowser -t "https://www.python.org"
```

Availability : not WASI, not Android.

L'exception suivante est définie :

exception webbrowser.**Error**

Exception raised when a browser control error occurs.

Les fonctions suivantes sont définies :

webbrowser.**open**(*url*, *new=0*, *autoraise=True*)

Display *url* using the default browser. If *new* is 0, the *url* is opened in the same browser window if possible. If *new* is 1, a new browser window is opened if possible. If *new* is 2, a new browser page ("tab") is opened if possible. If *autoraise* is `True`, the window is raised if possible (note that under many window managers this will occur regardless of the setting of this variable).

Returns `True` if a browser was successfully launched, `False` otherwise.

Note that on some platforms, trying to open a filename using this function, may work and start the operating system's associated program. However, this is neither supported nor portable.

Raises an *auditing event* `webbrowser.open` with argument `url`.

webbrowser.**open_new**(*url*)

Open *url* in a new window of the default browser, if possible, otherwise, open *url* in the only browser window.

Returns `True` if a browser was successfully launched, `False` otherwise.

webbrowser.**open_new_tab**(*url*)

Open *url* in a new page ("tab") of the default browser, if possible, otherwise equivalent to `open_new()`.

Returns `True` if a browser was successfully launched, `False` otherwise.

webbrowser.**get**(*using=None*)

Return a controller object for the browser type *using*. If *using* is `None`, return a controller for a default browser appropriate to the caller's environment.

webbrowser.**register**(*name*, *constructor*, *instance=None*, *, *preferred=False*)

Register the browser type *name*. Once a browser type is registered, the `get()` function can return a controller for that browser type. If *instance* is not provided, or is `None`, *constructor* will be called without parameters to create an instance when needed. If *instance* is provided, *constructor* will never be called, and may be `None`.

Setting *preferred* to `True` makes this browser a preferred result for a `get()` call with no argument. Otherwise, this entry point is only useful if you plan to either set the `BROWSER` variable or call `get()` with a nonempty argument matching the name of a handler you declare.

Modifié dans la version 3.7 : *preferred* keyword-only parameter was added.

A number of browser types are predefined. This table gives the type names that may be passed to the `get()` function and the corresponding instantiations for the controller classes, all defined in this module.

Type Name	Class Name	Notes
`'mozilla'`	`Mozilla('mozilla')`	
`'firefox'`	`Mozilla('mozilla')`	
`'epiphany'`	`Epiphany('epiphany')`	
`'kfmclient'`	`Konqueror()`	(1)
`'konqueror'`	`Konqueror()`	(1)
`'kfm'`	`Konqueror()`	(1)
`'opera'`	`Opera()`	
`'links'`	`GenericBrowser('links')`	
`'elinks'`	`Elinks('elinks')`	
`'lynx'`	`GenericBrowser('lynx')`	
`'w3m'`	`GenericBrowser('w3m')`	
`'windows-default'`	`WindowsDefault`	(2)
`'macosx'`	`MacOSXOSAScript('default')`	(3)
`'safari'`	`MacOSXOSAScript('safari')`	(3)
`'google-chrome'`	`Chrome('google-chrome')`	
`'chrome'`	`Chrome('chrome')`	
`'chromium'`	`Chromium('chromium')`	
`'chromium-browser'`	`Chromium('chromium-browser')`	
`'iosbrowser'`	`IOSBrowser`	(4)

Notes :

(1) "Konqueror" is the file manager for the KDE desktop environment for Unix, and only makes sense to use if KDE is running. Some way of reliably detecting KDE would be nice ; the `KDEDIR` variable is not sufficient. Note also that the name "kfm" is used even when using the **`konqueror`** command with KDE 2 --- the implementation selects the best strategy for running Konqueror.

(2) Only on Windows platforms.

(3) Only on macOS.

(4) Only on iOS.

Ajouté dans la version 3.2 : A new `MacOSXOSAScript` class has been added and is used on Mac instead of the previous `MacOSX` class. This adds support for opening browsers not currently set as the OS default.

Ajouté dans la version 3.3 : Support for Chrome/Chromium has been added.

Modifié dans la version 3.12 : Support for several obsolete browsers has been removed. Removed browsers include Grail, Mosaic, Netscape, Galeon, Skipstone, Iceape, and Firefox versions 35 and below.

Modifié dans la version 3.13 : Support for iOS has been added.

Here are some simple examples :

```
url = 'https://docs.python.org/'

# Open URL in a new tab, if a browser window is already open.
webbrowser.open_new_tab(url)

# Open URL in new window, raising the window if possible.
webbrowser.open_new(url)
```

22.1.1 Browser Controller Objects

Browser controllers provide these methods which parallel three of the module-level convenience functions :

`controller.`**`name`**

System-dependent name for the browser.

`controller.open(url, new=0, autoraise=True)`

Display *url* using the browser handled by this controller. If *new* is 1, a new browser window is opened if possible. If *new* is 2, a new browser page ("tab") is opened if possible.

`controller.open_new(url)`

Open *url* in a new window of the browser handled by this controller, if possible, otherwise, open *url* in the only browser window. Alias `open_new()`.

`controller.open_new_tab(url)`

Open *url* in a new page ("tab") of the browser handled by this controller, if possible, otherwise equivalent to `open_new()`.

Notes

22.2 `wsgiref` --- WSGI Utilities and Reference Implementation

Source code : Lib/wsgiref

WSGI (*Web Server Gateway Interface*) est une interface standard entre le serveur web et une application web écrite en Python. Avoir une interface standardisée permet de faciliter l'usage de ces applications avec un certain nombre de serveurs web différents.

Seules les personnes programmant des serveurs web et des cadriciels ont besoin de connaître les détails d'implémentation et les cas particuliers de l'architecture de WSGI. En tant qu'utilisateur WSGI vous avez uniquement besoin d'installer WSGI ou d'utiliser un cadriciel existant.

`wsgiref` is a reference implementation of the WSGI specification that can be used to add WSGI support to a web server or framework. It provides utilities for manipulating WSGI environment variables and response headers, base classes for implementing WSGI servers, a demo HTTP server that serves WSGI applications, types for static type checking, and a validation tool that checks WSGI servers and applications for conformance to the WSGI specification (**PEP 3333**).

Voir wsgi.readthedocs.io pour plus d'informations à propos de WSGI ainsi que des liens vers des tutoriels et d'autres ressources.

22.2.1 `wsgiref.util` — outils pour les environnements WSGI

This module provides a variety of utility functions for working with WSGI environments. A WSGI environment is a dictionary containing HTTP request variables as described in **PEP 3333**. All of the functions taking an *environ* parameter expect a WSGI-compliant dictionary to be supplied ; please see **PEP 3333** for a detailed specification and `WSGIEnvironment` for a type alias that can be used in type annotations.

`wsgiref.util.guess_scheme(environ)`

Tente de déterminer s'il faut assigner "http" ou "https" à `wsgi.url_sheme`, en vérifiant si une variable d'environnement `HTTPS` est dans le dictionnaire *environ*. La valeur renvoyée est une chaîne de caractères.

This function is useful when creating a gateway that wraps CGI or a CGI-like protocol such as FastCGI. Typically, servers providing such protocols will include a `HTTPS` variable with a value of "1", "yes", or "on" when a request is received via SSL. So, this function returns "https" if such a value is found, and "http" otherwise.

`wsgiref.util.request_uri(environ, include_query=True)`

Return the full request URI, optionally including the query string, using the algorithm found in the "URL Reconstruction" section of **PEP 3333**. If *include_query* is false, the query string is not included in the resulting URI.

`wsgiref.util.application_uri(environ)`

Similar to `request_uri()`, except that the `PATH_INFO` and `QUERY_STRING` variables are ignored. The result is the base URI of the application object addressed by the request.

wsgiref.util.**shift_path_info**(*environ*)

Shift a single name from PATH_INFO to SCRIPT_NAME and return the name. The *environ* dictionary is *modified* in-place ; use a copy if you need to keep the original PATH_INFO or SCRIPT_NAME intact.

If there are no remaining path segments in PATH_INFO, None is returned.

Typically, this routine is used to process each portion of a request URI path, for example to treat the path as a series of dictionary keys. This routine modifies the passed-in environment to make it suitable for invoking another WSGI application that is located at the target URI. For example, if there is a WSGI application at /foo, and the request URI path is /foo/bar/baz, and the WSGI application at /foo calls *shift_path_info()*, it will receive the string "bar", and the environment will be updated to be suitable for passing to a WSGI application at /foo/bar. That is, SCRIPT_NAME will change from /foo to /foo/bar, and PATH_INFO will change from /bar/baz to /baz.

When PATH_INFO is just a "/", this routine returns an empty string and appends a trailing slash to SCRIPT_NAME, even though empty path segments are normally ignored, and SCRIPT_NAME doesn't normally end in a slash. This is intentional behavior, to ensure that an application can tell the difference between URIs ending in /x from ones ending in /x/ when using this routine to do object traversal.

wsgiref.util.**setup_testing_defaults**(*environ*)

Met à jour *environ* avec des valeurs par défaut pour des cas de tests.

Cette fonction ajoute des paramètres requis pour WSGI, en particulier HTTP_HOST, SERVER_NAME, SERVER_PORT, REQUEST_METHOD, SCRIPT_NAME, PATH_INFO et toutes les autres variables WSGI définies dans la **PEP 3333**. Elle ne fournit que des valeurs par défaut sans toucher aux valeurs déjà définies de ces variables.

Cette fonction a pour but de faciliter les tests unitaires des serveurs et des applications WSGI dans des environnements factices. Elle ne devrait pas être utilisée dans une application ou un serveur WSGI, étant donné que les données sont factices !

Exemple d'utilisation :

```
from wsgiref.util import setup_testing_defaults
from wsgiref.simple_server import make_server

# A relatively simple WSGI application. It's going to print out the
# environment dictionary after being updated by setup_testing_defaults
def simple_app(environ, start_response):
    setup_testing_defaults(environ)

    status = '200 OK'
    headers = [('Content-type', 'text/plain; charset=utf-8')]

    start_response(status, headers)

    ret = [("%s: %s\n" % (key, value)).encode("utf-8")
           for key, value in environ.items()]
    return ret

with make_server('', 8000, simple_app) as httpd:
    print("Serving on port 8000...")
    httpd.serve_forever()
```

In addition to the environment functions above, the *wsgiref.util* module also provides these miscellaneous utilities :

wsgiref.util.**is_hop_by_hop**(*header_name*)

Return True if 'header_name' is an HTTP/1.1 "Hop-by-Hop" header, as defined by **RFC 2616**.

class wsgiref.util.**FileWrapper**(*filelike*, *blksize=8192*)

A concrete implementation of the *wsgiref.types.FileWrapper* protocol used to convert a file-like object

to an *iterator*. The resulting objects are *iterable*s. As the object is iterated over, the optional *blksize* parameter will be repeatedly passed to the *filelike* object's `read()` method to obtain bytestrings to yield. When `read()` returns an empty bytestring, iteration is ended and is not resumable.

If *filelike* has a `close()` method, the returned object will also have a `close()` method, and it will invoke the *filelike* object's `close()` method when called.

Exemple d'utilisation :

```
from io import StringIO
from wsgiref.util import FileWrapper

# We're using a StringIO-buffer for as the file-like object
filelike = StringIO("This is an example file-like object"*10)
wrapper = FileWrapper(filelike, blksize=5)

for chunk in wrapper:
    print(chunk)
```

Modifié dans la version 3.11 : Support for `__getitem__()` method has been removed.

22.2.2 `wsgiref.headers` -- WSGI response header tools

This module provides a single class, `Headers`, for convenient manipulation of WSGI response headers using a mapping-like interface.

class `wsgiref.headers.`**`Headers`**([*headers*])

Create a mapping-like object wrapping *headers*, which must be a list of header name/value tuples as described in **PEP 3333**. The default value of *headers* is an empty list.

`Headers` objects support typical mapping operations including `__getitem__()`, `get()`, `__setitem__()`, `setdefault()`, `__delitem__()` and `__contains__()`. For each of these methods, the key is the header name (treated case-insensitively), and the value is the first value associated with that header name. Setting a header deletes any existing values for that header, then adds a new value at the end of the wrapped header list. Headers' existing order is generally maintained, with new headers added to the end of the wrapped list.

Unlike a dictionary, `Headers` objects do not raise an error when you try to get or delete a key that isn't in the wrapped header list. Getting a nonexistent header just returns `None`, and deleting a nonexistent header does nothing.

`Headers` objects also support `keys()`, `values()`, and `items()` methods. The lists returned by `keys()` and `items()` can include the same key more than once if there is a multi-valued header. The `len()` of a `Headers` object is the same as the length of its `items()`, which is the same as the length of the wrapped header list. In fact, the `items()` method just returns a copy of the wrapped header list.

Calling `bytes()` on a `Headers` object returns a formatted bytestring suitable for transmission as HTTP response headers. Each header is placed on a line with its value, separated by a colon and a space. Each line is terminated by a carriage return and line feed, and the bytestring is terminated with a blank line.

In addition to their mapping interface and formatting features, `Headers` objects also have the following methods for querying and adding multi-valued headers, and for adding headers with MIME parameters :

`get_all`(*name*)

Renvoie une liste de toutes les valeurs pour l'en-tête *name*.

The returned list will be sorted in the order they appeared in the original header list or were added to this instance, and may contain duplicates. Any fields deleted and re-inserted are always appended to the header list. If no fields exist with the given name, returns an empty list.

`add_header`(*name*, *value*, ***_params*)

Add a (possibly multi-valued) header, with optional MIME parameters specified via keyword arguments.

name is the header field to add. Keyword arguments can be used to set MIME parameters for the header field. Each parameter must be a string or `None`. Underscores in parameter names are converted to dashes,

since dashes are illegal in Python identifiers, but many MIME parameter names include dashes. If the parameter value is a string, it is added to the header value parameters in the form `name="value"`. If it is `None`, only the parameter name is added. (This is used for MIME parameters without a value.) Example usage :

```
h.add_header('content-disposition', 'attachment', filename='bud.gif')
```

Le code ci-dessus ajoute un en-tête qui ressemble à ceci :

```
Content-Disposition: attachment; filename="bud.gif"
```

Modifié dans la version 3.5 : Le paramètre *headers* est optionnel.

22.2.3 `wsgiref.simple_server` -- a simple WSGI HTTP server

This module implements a simple HTTP server (based on `http.server`) that serves WSGI applications. Each server instance serves a single WSGI application on a given host and port. If you want to serve multiple applications on a single host and port, you should create a WSGI application that parses `PATH_INFO` to select which application to invoke for each request. (E.g., using the `shift_path_info()` function from `wsgiref.util`.)

`wsgiref.simple_server.`**`make_server`**(*host*, *port*, *app*, *server_class=WSGIServer*, *handler_class=WSGIRequestHandler*)

Create a new WSGI server listening on *host* and *port*, accepting connections for *app*. The return value is an instance of the supplied *server_class*, and will process requests using the specified *handler_class*. *app* must be a WSGI application object, as defined by **PEP 3333**.

Exemple d'utilisation :

```
from wsgiref.simple_server import make_server, demo_app

with make_server('', 8000, demo_app) as httpd:
    print("Serving HTTP on port 8000...")

    # Respond to requests until process is killed
    httpd.serve_forever()

    # Alternative: serve one request, then exit
    httpd.handle_request()
```

`wsgiref.simple_server.`**`demo_app`**(*environ*, *start_response*)

This function is a small but complete WSGI application that returns a text page containing the message "Hello world !" and a list of the key/value pairs provided in the *environ* parameter. It's useful for verifying that a WSGI server (such as `wsgiref.simple_server`) is able to run a simple WSGI application correctly.

class `wsgiref.simple_server.`**`WSGIServer`**(*server_address*, *RequestHandlerClass*)

Create a `WSGIServer` instance. *server_address* should be a `(host,port)` tuple, and *RequestHandlerClass* should be the subclass of `http.server.BaseHTTPRequestHandler` that will be used to process requests.

You do not normally need to call this constructor, as the `make_server()` function can handle all the details for you.

`WSGIServer` is a subclass of `http.server.HTTPServer`, so all of its methods (such as `serve_forever()` and `handle_request()`) are available. `WSGIServer` also provides these WSGI-specific methods :

`set_app`(*application*)

Sets the callable *application* as the WSGI application that will receive requests.

`get_app`()

Returns the currently set application callable.

Normally, however, you do not need to use these additional methods, as `set_app()` is normally called by `make_server()`, and the `get_app()` exists mainly for the benefit of request handler instances.

class wsgiref.simple_server.**WSGIRequestHandler**(*request*, *client_address*, *server*)

Create an HTTP handler for the given *request* (i.e. a socket), *client_address* (a `(host,port)` tuple), and *server* (`WSGIServer` instance).

You do not need to create instances of this class directly ; they are automatically created as needed by `WSGIServer` objects. You can, however, subclass this class and supply it as a *handler_class* to the `make_server()` function. Some possibly relevant methods for overriding in subclasses :

get_environ()

Return a `WSGIEnvironment` dictionary for a request. The default implementation copies the contents of the `WSGIServer` object's `base_environ` dictionary attribute and then adds various headers derived from the HTTP request. Each call to this method should return a new dictionary containing all of the relevant CGI environment variables as specified in **PEP 3333**.

get_stderr()

Return the object that should be used as the `wsgi.errors` stream. The default implementation just returns `sys.stderr`.

handle()

Process the HTTP request. The default implementation creates a handler instance using a `wsgiref.handlers` class to implement the actual WSGI application interface.

22.2.4 `wsgiref.validate` --- WSGI conformance checker

When creating new WSGI application objects, frameworks, servers, or middleware, it can be useful to validate the new code's conformance using `wsgiref.validate`. This module provides a function that creates WSGI application objects that validate communications between a WSGI server or gateway and a WSGI application object, to check both sides for protocol conformance.

Note that this utility does not guarantee complete **PEP 3333** compliance ; an absence of errors from this module does not necessarily mean that errors do not exist. However, if this module does produce an error, then it is virtually certain that either the server or application is not 100% compliant.

This module is based on the `paste.lint` module from Ian Bicking's "Python Paste" library.

wsgiref.validate.**validator**(*application*)

Wrap *application* and return a new WSGI application object. The returned application will forward all requests to the original *application*, and will check that both the *application* and the server invoking it are conforming to the WSGI specification and to **RFC 2616**.

Any detected nonconformance results in an `AssertionError` being raised ; note, however, that how these errors are handled is server-dependent. For example, `wsgiref.simple_server` and other servers based on `wsgiref.handlers` (that don't override the error handling methods to do something else) will simply output a message that an error has occurred, and dump the traceback to `sys.stderr` or some other error stream.

This wrapper may also generate output using the `warnings` module to indicate behaviors that are questionable but which may not actually be prohibited by **PEP 3333**. Unless they are suppressed using Python command-line options or the `warnings` API, any such warnings will be written to `sys.stderr` (*not* `wsgi.errors`, unless they happen to be the same object).

Exemple d'utilisation :

```
from wsgiref.validate import validator
from wsgiref.simple_server import make_server

# Our callable object which is intentionally not compliant to the
# standard, so the validator is going to break
def simple_app(environ, start_response):
    status = '200 OK'  # HTTP Status
    headers = [('Content-type', 'text/plain')]  # HTTP Headers
    start_response(status, headers)
```

(suite sur la page suivante)

(suite de la page précédente)

```
    # This is going to break because we need to return a list, and
    # the validator is going to inform us
    return b"Hello World"

# This is the application wrapped in a validator
validator_app = validator(simple_app)

with make_server('', 8000, validator_app) as httpd:
    print("Listening on port 8000....")
    httpd.serve_forever()
```

22.2.5 wsgiref.handlers -- server/gateway base classes

This module provides base handler classes for implementing WSGI servers and gateways. These base classes handle most of the work of communicating with a WSGI application, as long as they are given a CGI-like environment, along with input, output, and error streams.

class wsgiref.handlers.**CGIHandler**

CGI-based invocation via sys.stdin, sys.stdout, sys.stderr and os.environ. This is useful when you have a WSGI application and want to run it as a CGI script. Simply invoke CGIHandler().run(app), where app is the WSGI application object you wish to invoke.

This class is a subclass of *BaseCGIHandler* that sets wsgi.run_once to true, wsgi.multithread to false, and wsgi.multiprocess to true, and always uses *sys* and *os* to obtain the necessary CGI streams and environment.

class wsgiref.handlers.**IISCGIHandler**

A specialized alternative to *CGIHandler*, for use when deploying on Microsoft's IIS web server, without having set the config allowPathInfo option (IIS>=7) or metabase allowPathInfoForScriptMappings (IIS<7).

By default, IIS gives a PATH_INFO that duplicates the SCRIPT_NAME at the front, causing problems for WSGI applications that wish to implement routing. This handler strips any such duplicated path.

IIS can be configured to pass the correct PATH_INFO, but this causes another bug where PATH_TRANSLATED is wrong. Luckily this variable is rarely used and is not guaranteed by WSGI. On IIS<7, though, the setting can only be made on a vhost level, affecting all other script mappings, many of which break when exposed to the PATH_TRANSLATED bug. For this reason IIS<7 is almost never deployed with the fix (Even IIS7 rarely uses it because there is still no UI for it.).

There is no way for CGI code to tell whether the option was set, so a separate handler class is provided. It is used in the same way as *CGIHandler*, i.e., by calling IISCGIHandler().run(app), where app is the WSGI application object you wish to invoke.

Ajouté dans la version 3.2.

class wsgiref.handlers.**BaseCGIHandler**(*stdin*, *stdout*, *stderr*, *environ*, *multithread=True*, *multiprocess=False*)

Similar to *CGIHandler*, but instead of using the *sys* and *os* modules, the CGI environment and I/O streams are specified explicitly. The *multithread* and *multiprocess* values are used to set the wsgi.multithread and wsgi.multiprocess flags for any applications run by the handler instance.

This class is a subclass of *SimpleHandler* intended for use with software other than HTTP "origin servers". If you are writing a gateway protocol implementation (such as CGI, FastCGI, SCGI, etc.) that uses a Status: header to send an HTTP status, you probably want to subclass this instead of *SimpleHandler*.

class wsgiref.handlers.**SimpleHandler**(*stdin*, *stdout*, *stderr*, *environ*, *multithread=True*, *multiprocess=False*)

Similar to *BaseCGIHandler*, but designed for use with HTTP origin servers. If you are writing an HTTP server implementation, you will probably want to subclass this instead of *BaseCGIHandler*.

This class is a subclass of *BaseHandler*. It overrides the __init__(), *get_stdin()*, *get_stderr()*,

add_cgi_vars(), *_write()*, and *_flush()* methods to support explicitly setting the environment and streams via the constructor. The supplied environment and streams are stored in the `stdin`, `stdout`, `stderr`, and `environ` attributes.

The *write()* method of *stdout* should write each chunk in full, like *io.BufferedIOBase*.

class wsgiref.handlers.**BaseHandler**

This is an abstract base class for running WSGI applications. Each instance will handle a single HTTP request, although in principle you could create a subclass that was reusable for multiple requests.

BaseHandler instances have only one method intended for external use :

run(*app*)

Run the specified WSGI application, *app*.

All of the other *BaseHandler* methods are invoked by this method in the process of running the application, and thus exist primarily to allow customizing the process.

The following methods MUST be overridden in a subclass :

_write(*data*)

Buffer the bytes *data* for transmission to the client. It's okay if this method actually transmits the data ; *BaseHandler* just separates write and flush operations for greater efficiency when the underlying system actually has such a distinction.

_flush()

Force buffered data to be transmitted to the client. It's okay if this method is a no-op (i.e., if *_write()* actually sends the data).

get_stdin()

Return an object compatible with *InputStream* suitable for use as the `wsgi.input` of the request currently being processed.

get_stderr()

Return an object compatible with *ErrorStream* suitable for use as the `wsgi.errors` of the request currently being processed.

add_cgi_vars()

Insert CGI variables for the current request into the `environ` attribute.

Here are some other methods and attributes you may wish to override. This list is only a summary, however, and does not include every method that can be overridden. You should consult the docstrings and source code for additional information before attempting to create a customized *BaseHandler* subclass.

Attributes and methods for customizing the WSGI environment :

wsgi_multithread

The value to be used for the `wsgi.multithread` environment variable. It defaults to true in *BaseHandler*, but may have a different default (or be set by the constructor) in the other subclasses.

wsgi_multiprocess

The value to be used for the `wsgi.multiprocess` environment variable. It defaults to true in *BaseHandler*, but may have a different default (or be set by the constructor) in the other subclasses.

wsgi_run_once

The value to be used for the `wsgi.run_once` environment variable. It defaults to false in *BaseHandler*, but *CGIHandler* sets it to true by default.

os_environ

The default environment variables to be included in every request's WSGI environment. By default, this is a copy of `os.environ` at the time that *wsgiref.handlers* was imported, but subclasses can either create their own at the class or instance level. Note that the dictionary should be considered read-only, since the default value is shared between multiple classes and instances.

server_software

If the *origin_server* attribute is set, this attribute's value is used to set the default `SERVER_SOFTWARE` WSGI environment variable, and also to set a default `Server:` header in HTTP responses. It is ignored for handlers (such as *BaseCGIHandler* and *CGIHandler*) that are not HTTP origin servers.

Modifié dans la version 3.3 : The term "Python" is replaced with implementation specific term like "CPython", "Jython" etc.

get_scheme()

Return the URL scheme being used for the current request. The default implementation uses the `guess_scheme()` function from *wsgiref.util* to guess whether the scheme should be "http" or "https", based on the current request's `environ` variables.

setup_environ()

Set the `environ` attribute to a fully populated WSGI environment. The default implementation uses all of the above methods and attributes, plus the *get_stdin()*, *get_stderr()*, and *add_cgi_vars()* methods and the *wsgi_file_wrapper* attribute. It also inserts a `SERVER_SOFTWARE` key if not present, as long as the *origin_server* attribute is a true value and the *server_software* attribute is set.

Methods and attributes for customizing exception handling :

log_exception(*exc_info*)

Log the *exc_info* tuple in the server log. *exc_info* is a `(type, value, traceback)` tuple. The default implementation simply writes the traceback to the request's `wsgi.errors` stream and flushes it. Subclasses can override this method to change the format or retarget the output, mail the traceback to an administrator, or whatever other action may be deemed suitable.

traceback_limit

The maximum number of frames to include in tracebacks output by the default *log_exception()* method. If `None`, all frames are included.

error_output(*environ*, *start_response*)

This method is a WSGI application to generate an error page for the user. It is only invoked if an error occurs before headers are sent to the client.

This method can access the current error using `sys.exception()`, and should pass that information to *start_response* when calling it (as described in the "Error Handling" section of **PEP 3333**).

The default implementation just uses the *error_status*, *error_headers*, and *error_body* attributes to generate an output page. Subclasses can override this to produce more dynamic error output.

Note, however, that it's not recommended from a security perspective to spit out diagnostics to any old user ; ideally, you should have to do something special to enable diagnostic output, which is why the default implementation doesn't include any.

error_status

The HTTP status used for error responses. This should be a status string as defined in **PEP 3333** ; it defaults to a 500 code and message.

error_headers

The HTTP headers used for error responses. This should be a list of WSGI response headers (`(name, value)` tuples), as described in **PEP 3333**. The default list just sets the content type to `text/plain`.

error_body

The error response body. This should be an HTTP response body bytestring. It defaults to the plain text, "A server error occurred. Please contact the administrator."

Methods and attributes for **PEP 3333**'s "Optional Platform-Specific File Handling" feature :

wsgi_file_wrapper

A `wsgi.file_wrapper` factory, compatible with *wsgiref.types.FileWrapper*, or `None`. The default value of this attribute is the *wsgiref.util.FileWrapper* class.

sendfile()

Override to implement platform-specific file transmission. This method is called only if the application's return value is an instance of the class specified by the `wsgi_file_wrapper` attribute. It should return a true value if it was able to successfully transmit the file, so that the default transmission code will not be executed. The default implementation of this method just returns a false value.

Miscellaneous methods and attributes :

origin_server

This attribute should be set to a true value if the handler's `_write()` and `_flush()` are being used to communicate directly to the client, rather than via a CGI-like gateway protocol that wants the HTTP status in a special `Status:` header.

This attribute's default value is true in `BaseHandler`, but false in `BaseCGIHandler` and `CGIHandler`.

http_version

If `origin_server` is true, this string attribute is used to set the HTTP version of the response set to the client. It defaults to `"1.0"`.

`wsgiref.handlers.`**read_environ**`()`

Transcode CGI variables from `os.environ` to **PEP 3333** "bytes in unicode" strings, returning a new dictionary. This function is used by `CGIHandler` and `IISCGIHandler` in place of directly using `os.environ`, which is not necessarily WSGI-compliant on all platforms and web servers using Python 3 -- specifically, ones where the OS's actual environment is Unicode (i.e. Windows), or ones where the environment is bytes, but the system encoding used by Python to decode it is anything other than ISO-8859-1 (e.g. Unix systems using UTF-8).

If you are implementing a CGI-based handler of your own, you probably want to use this routine instead of just copying values out of `os.environ` directly.

Ajouté dans la version 3.2.

22.2.6 `wsgiref.types` -- WSGI types for static type checking

This module provides various types for static type checking as described in **PEP 3333**.

Ajouté dans la version 3.11.

class `wsgiref.types.`**StartResponse**

A `typing.Protocol` describing **start_response()** callables (**PEP 3333**).

`wsgiref.types.`**WSGIEnvironment**

A type alias describing a WSGI environment dictionary.

`wsgiref.types.`**WSGIApplication**

A type alias describing a WSGI application callable.

class `wsgiref.types.`**InputStream**

A `typing.Protocol` describing a **WSGI Input Stream**.

class `wsgiref.types.`**ErrorStream**

A `typing.Protocol` describing a **WSGI Error Stream**.

class `wsgiref.types.`**FileWrapper**

A `typing.Protocol` describing a **file wrapper**. See `wsgiref.util.FileWrapper` for a concrete implementation of this protocol.

22.2.7 Exemples

This is a working "Hello World" WSGI application :

```
"""
Every WSGI application must have an application object - a callable
object that accepts two arguments. For that purpose, we're going to
use a function (note that you're not limited to a function, you can
use a class for example). The first argument passed to the function
is a dictionary containing CGI-style environment variables and the
second variable is the callable object.
"""
from wsgiref.simple_server import make_server


def hello_world_app(environ, start_response):
    status = "200 OK"  # HTTP Status
    headers = [("Content-type", "text/plain; charset=utf-8")]  # HTTP Headers
    start_response(status, headers)

    # The returned object is going to be printed
    return [b"Hello World"]

with make_server("", 8000, hello_world_app) as httpd:
    print("Serving on port 8000...")

    # Serve until process is killed
    httpd.serve_forever()
```

Example of a WSGI application serving the current directory, accept optional directory and port number (default : 8000) on the command line :

```
"""
Small wsgiref based web server. Takes a path to serve from and an
optional port number (defaults to 8000), then tries to serve files.
MIME types are guessed from the file names, 404 errors are raised
if the file is not found.
"""
import mimetypes
import os
import sys
from wsgiref import simple_server, util


def app(environ, respond):
    # Get the file name and MIME type
    fn = os.path.join(path, environ["PATH_INFO"][1:])
    if "." not in fn.split(os.path.sep)[-1]:
        fn = os.path.join(fn, "index.html")
    mime_type = mimetypes.guess_file_type(fn)[0]

    # Return 200 OK if file exists, otherwise 404 Not Found
    if os.path.exists(fn):
        respond("200 OK", [("Content-Type", mime_type)])
        return util.FileWrapper(open(fn, "rb"))
    else:
        respond("404 Not Found", [("Content-Type", "text/plain")])
        return [b"not found"]


if __name__ == "__main__":
```

(suite sur la page suivante)

(suite de la page précédente)

```
    # Get the path and port from command-line arguments
    path = sys.argv[1] if len(sys.argv) > 1 else os.getcwd()
    port = int(sys.argv[2]) if len(sys.argv) > 2 else 8000

    # Make and start the server until control-c
    httpd = simple_server.make_server("", port, app)
    print(f"Serving {path} on port {port}, control-C to stop")
    try:
        httpd.serve_forever()
    except KeyboardInterrupt:
        print("Shutting down.")
        httpd.server_close()
```

22.3 `urllib` --- URL handling modules

Code source : Lib/urllib/

`urllib` est un paquet qui collecte plusieurs modules travaillant avec les URLs :

- *`urllib.request`* pour ouvrir et lire des URLs ;
- *`urllib.error`* contenant les exceptions levées par *`urllib.request`* ;
- *`urllib.parse`* pour analyser les URLs ;
- *`urllib.robotparser`* pour analyser les fichiers `robots.txt`.

22.4 `urllib.request` --- Extensible library for opening URLs

Source code : Lib/urllib/request.py

The *`urllib.request`* module defines functions and classes which help in opening URLs (mostly HTTP) in a complex world --- basic and digest authentication, redirections, cookies and more.

Voir aussi

The Requests package is recommended for a higher-level HTTP client interface.

Avertissement

On macOS it is unsafe to use this module in programs using *`os.fork()`* because the *`getproxies()`* implementation for macOS uses a higher-level system API. Set the environment variable `no_proxy` to `*` to avoid this problem (e.g. `os.environ["no_proxy"] = "*"`).

Availability : not WASI.

This module does not work or is not available on WebAssembly. See *Plateformes WebAssembly* for more information.

The *`urllib.request`* module defines the following functions :

`urllib.request.`**`urlopen`**(*url*, *data=None*, [*timeout*,]*, *context=None*)

Open *url*, which can be either a string containing a valid, properly encoded URL, or a *`Request`* object.

data must be an object specifying additional data to be sent to the server, or `None` if no such data is needed. See *`Request`* for details.

urllib.request module uses HTTP/1.1 and includes `Connection:close` header in its HTTP requests.

The optional *timeout* parameter specifies a timeout in seconds for blocking operations like the connection attempt (if not specified, the global default timeout setting will be used). This actually only works for HTTP, HTTPS and FTP connections.

If *context* is specified, it must be a `ssl.SSLContext` instance describing the various SSL options. See `HTTPSConnection` for more details.

This function always returns an object which can work as a *context manager* and has the properties *url*, *headers*, and *status*. See `urllib.response.addinfourl` for more detail on these properties.

For HTTP and HTTPS URLs, this function returns a `http.client.HTTPResponse` object slightly modified. In addition to the three new methods above, the msg attribute contains the same information as the `reason` attribute --- the reason phrase returned by server --- instead of the response headers as it is specified in the documentation for `HTTPResponse`.

For FTP, file, and data URLs, this function returns a `urllib.response.addinfourl` object.

Raises `URLError` on protocol errors.

Note that `None` may be returned if no handler handles the request (though the default installed global `OpenerDirector` uses `UnknownHandler` to ensure this never happens).

In addition, if proxy settings are detected (for example, when a `*_proxy` environment variable like `http_proxy` is set), `ProxyHandler` is default installed and makes sure the requests are handled through the proxy.

The legacy `urllib.urlopen` function from Python 2.6 and earlier has been discontinued; `urllib.request.urlopen()` corresponds to the old `urllib2.urlopen`. Proxy handling, which was done by passing a dictionary parameter to `urllib.urlopen`, can be obtained by using `ProxyHandler` objects.

The default opener raises an *auditing event* `urllib.Request` with arguments `fullurl`, `data`, `headers`, `method` taken from the request object.

Modifié dans la version 3.2 : *cafile* and *capath* were added.

HTTPS virtual hosts are now supported if possible (that is, if `ssl.HAS_SNI` is true).

data can be an iterable object.

Modifié dans la version 3.3 : *cadefault* was added.

Modifié dans la version 3.4.3 : *context* was added.

Modifié dans la version 3.10 : HTTPS connection now send an ALPN extension with protocol indicator `http/1.1` when no *context* is given. Custom *context* should set ALPN protocols with `set_alpn_protocols()`.

Modifié dans la version 3.13 : Remove *cafile*, *capath* and *cadefault* parameters : use the *context* parameter instead.

`urllib.request.`**`install_opener`**(*opener*)

Install an `OpenerDirector` instance as the default global opener. Installing an opener is only necessary if you want urlopen to use that opener; otherwise, simply call `OpenerDirector.open()` instead of `urlopen()`. The code does not check for a real `OpenerDirector`, and any class with the appropriate interface will work.

`urllib.request.`**`build_opener`**([*handler*, ...])

Return an `OpenerDirector` instance, which chains the handlers in the order given. *handler*s can be either instances of `BaseHandler`, or subclasses of `BaseHandler` (in which case it must be possible to call the constructor without any parameters). Instances of the following classes will be in front of the *handler*s, unless the *handler*s contain them, instances of them or subclasses of them : `ProxyHandler` (if proxy settings are detected), `UnknownHandler`, `HTTPHandler`, `HTTPDefaultErrorHandler`, `HTTPRedirectHandler`, `FTPHandler`, `FileHandler`, `HTTPErrorProcessor`.

If the Python installation has SSL support (i.e., if the `ssl` module can be imported), `HTTPSHandler` will also be added.

A *BaseHandler* subclass may also change its `handler_order` attribute to modify its position in the handlers list.

`urllib.request.`**`pathname2url`**(*path*)

Convert the given local path to a `file:` URL. This function uses *quote()* function to encode the path. For historical reasons, the return value omits the `file:` scheme prefix. This example shows the function being used on Windows :

```
>>> from urllib.request import pathname2url
>>> path = 'C:\\Program Files'
>>> 'file:' + pathname2url(path)
'file:///C:/Program%20Files'
```

Modifié dans la version 3.14 : Paths beginning with a slash are converted to URLs with authority sections. For example, the path `/etc/hosts` is converted to the URL `///etc/hosts`.

Modifié dans la version 3.14 : Windows drive letters are no longer converted to uppercase, and `:` characters not following a drive letter no longer cause an *OSError* exception to be raised on Windows.

`urllib.request.`**`url2pathname`**(*url*)

Convert the given `file:` URL to a local path. This function uses *unquote()* to decode the URL. For historical reasons, the given value *must* omit the `file:` scheme prefix. This example shows the function being used on Windows :

```
>>> from urllib.request import url2pathname
>>> url = 'file:///C:/Program%20Files'
>>> url2pathname(url.removeprefix('file:'))
'C:\\Program Files'
```

Modifié dans la version 3.14 : Windows drive letters are no longer converted to uppercase.

`urllib.request.`**`getproxies`**()

This helper function returns a dictionary of scheme to proxy server URL mappings. It scans the environment for variables named `<scheme>_proxy`, in a case insensitive approach, for all operating systems first, and when it cannot find it, looks for proxy information from System Configuration for macOS and Windows Systems Registry for Windows. If both lowercase and uppercase environment variables exist (and disagree), lowercase is preferred.

> **Note**
>
> If the environment variable `REQUEST_METHOD` is set, which usually indicates your script is running in a CGI environment, the environment variable `HTTP_PROXY` (uppercase `_PROXY`) will be ignored. This is because that variable can be injected by a client using the "Proxy :" HTTP header. If you need to use an HTTP proxy in a CGI environment, either use `ProxyHandler` explicitly, or make sure the variable name is in lowercase (or at least the `_proxy` suffix).

The following classes are provided :

`class` `urllib.request.`**`Request`**(*url*, *data=None*, *headers={}*, *origin_req_host=None*, *unverifiable=False*, *method=None*)

This class is an abstraction of a URL request.

url should be a string containing a valid, properly encoded URL.

data must be an object specifying additional data to send to the server, or `None` if no such data is needed. Currently HTTP requests are the only ones that use *data*. The supported object types include bytes, file-like objects, and iterables of bytes-like objects. If no `Content-Length` nor `Transfer-Encoding` header field has been provided, *HTTPHandler* will set these headers according to the type of *data*. `Content-Length` will be used to send bytes objects, while `Transfer-Encoding: chunked` as specified in **RFC 7230**, Section 3.3.1 will be used to send files and other iterables.

For an HTTP POST request method, *data* should be a buffer in the standard `application/x-www-form-urlencoded` format. The `urllib.parse.urlencode()` function takes a mapping or sequence of 2-tuples and returns an ASCII string in this format. It should be encoded to bytes before being used as the *data* parameter.

headers should be a dictionary, and will be treated as if `add_header()` was called with each key and value as arguments. This is often used to "spoof" the `User-Agent` header value, which is used by a browser to identify itself -- some HTTP servers only allow requests coming from common browsers as opposed to scripts. For example, Mozilla Firefox may identify itself as `"Mozilla/5.0 (X11; U; Linux i686) Gecko/20071127 Firefox/2.0.0.11"`, while `urllib`'s default user agent string is `"Python-urllib/2.6"` (on Python 2.6). All header keys are sent in camel case.

An appropriate `Content-Type` header should be included if the *data* argument is present. If this header has not been provided and *data* is not `None`, `Content-Type: application/x-www-form-urlencoded` will be added as a default.

The next two arguments are only of interest for correct handling of third-party HTTP cookies :

origin_req_host should be the request-host of the origin transaction, as defined by **RFC 2965**. It defaults to `http.cookiejar.request_host(self)`. This is the host name or IP address of the original request that was initiated by the user. For example, if the request is for an image in an HTML document, this should be the request-host of the request for the page containing the image.

unverifiable should indicate whether the request is unverifiable, as defined by **RFC 2965**. It defaults to `False`. An unverifiable request is one whose URL the user did not have the option to approve. For example, if the request is for an image in an HTML document, and the user had no option to approve the automatic fetching of the image, this should be true.

method should be a string that indicates the HTTP request method that will be used (e.g. `'HEAD'`). If provided, its value is stored in the `method` attribute and is used by `get_method()`. The default is `'GET'` if *data* is `None` or `'POST'` otherwise. Subclasses may indicate a different default method by setting the `method` attribute in the class itself.

> **Note**
>
> The request will not work as expected if the data object is unable to deliver its content more than once (e.g. a file or an iterable that can produce the content only once) and the request is retried for HTTP redirects or authentication. The *data* is sent to the HTTP server right away after the headers. There is no support for a 100-continue expectation in the library.

Modifié dans la version 3.3 : `Request.method` argument is added to the Request class.

Modifié dans la version 3.4 : Default `Request.method` may be indicated at the class level.

Modifié dans la version 3.6 : Do not raise an error if the `Content-Length` has not been provided and *data* is neither `None` nor a bytes object. Fall back to use chunked transfer encoding instead.

class `urllib.request.`**`OpenerDirector`**

The `OpenerDirector` class opens URLs via `BaseHandler`s chained together. It manages the chaining of handlers, and recovery from errors.

class `urllib.request.`**`BaseHandler`**

This is the base class for all registered handlers --- and handles only the simple mechanics of registration.

class `urllib.request.`**`HTTPDefaultErrorHandler`**

A class which defines a default handler for HTTP error responses ; all responses are turned into `HTTPError` exceptions.

class `urllib.request.`**`HTTPRedirectHandler`**

A class to handle redirections.

class `urllib.request.`**`HTTPCookieProcessor`**(*cookiejar=None*)

A class to handle HTTP Cookies.

class urllib.request.**ProxyHandler**(*proxies=None*)

Cause requests to go through a proxy. If *proxies* is given, it must be a dictionary mapping protocol names to URLs of proxies. The default is to read the list of proxies from the environment variables `<protocol>_proxy`. If no proxy environment variables are set, then in a Windows environment proxy settings are obtained from the registry's Internet Settings section, and in a macOS environment proxy information is retrieved from the System Configuration Framework.

To disable autodetected proxy pass an empty dictionary.

The `no_proxy` environment variable can be used to specify hosts which shouldn't be reached via proxy; if set, it should be a comma-separated list of hostname suffixes, optionally with `:port` appended, for example `cern.ch,ncsa.uiuc.edu,some.host:8080`.

> **Note**
>
> `HTTP_PROXY` will be ignored if a variable `REQUEST_METHOD` is set; see the documentation on *getproxies()*.

class urllib.request.**HTTPPasswordMgr**

Keep a database of `(realm, uri) -> (user, password)` mappings.

class urllib.request.**HTTPPasswordMgrWithDefaultRealm**

Keep a database of `(realm, uri) -> (user, password)` mappings. A realm of `None` is considered a catch-all realm, which is searched if no other realm fits.

class urllib.request.**HTTPPasswordMgrWithPriorAuth**

A variant of *HTTPPasswordMgrWithDefaultRealm* that also has a database of `uri -> is_authenticated` mappings. Can be used by a BasicAuth handler to determine when to send authentication credentials immediately instead of waiting for a `401` response first.

Ajouté dans la version 3.5.

class urllib.request.**AbstractBasicAuthHandler**(*password_mgr=None*)

This is a mixin class that helps with HTTP authentication, both to the remote host and to a proxy. *password_mgr*, if given, should be something that is compatible with *HTTPPasswordMgr*; refer to section *HTTPPasswordMgr Objects* for information on the interface that must be supported. If *passwd_mgr* also provides `is_authenticated` and `update_authenticated` methods (see *HTTPPasswordMgrWithPriorAuth Objects*), then the handler will use the `is_authenticated` result for a given URI to determine whether or not to send authentication credentials with the request. If `is_authenticated` returns `True` for the URI, credentials are sent. If `is_authenticated` is `False`, credentials are not sent, and then if a `401` response is received the request is re-sent with the authentication credentials. If authentication succeeds, `update_authenticated` is called to set `is_authenticated` `True` for the URI, so that subsequent requests to the URI or any of its super-URIs will automatically include the authentication credentials.

Ajouté dans la version 3.5 : Added `is_authenticated` support.

class urllib.request.**HTTPBasicAuthHandler**(*password_mgr=None*)

Handle authentication with the remote host. *password_mgr*, if given, should be something that is compatible with *HTTPPasswordMgr*; refer to section *HTTPPasswordMgr Objects* for information on the interface that must be supported. HTTPBasicAuthHandler will raise a *ValueError* when presented with a wrong Authentication scheme.

class urllib.request.**ProxyBasicAuthHandler**(*password_mgr=None*)

Handle authentication with the proxy. *password_mgr*, if given, should be something that is compatible with *HTTPPasswordMgr*; refer to section *HTTPPasswordMgr Objects* for information on the interface that must be supported.

class urllib.request.**AbstractDigestAuthHandler**(*password_mgr=None*)

This is a mixin class that helps with HTTP authentication, both to the remote host and to a proxy. *password_mgr*, if given, should be something that is compatible with *HTTPPasswordMgr*; refer to section *HTTPPasswordMgr Objects* for information on the interface that must be supported.

Modifié dans la version 3.14 : Added support for HTTP digest authentication algorithm `SHA-256`.

class `urllib.request.`**`HTTPDigestAuthHandler`**(*password_mgr=None*)

Handle authentication with the remote host. *password_mgr*, if given, should be something that is compatible with `HTTPPasswordMgr`; refer to section *HTTPPasswordMgr Objects* for information on the interface that must be supported. When both Digest Authentication Handler and Basic Authentication Handler are both added, Digest Authentication is always tried first. If the Digest Authentication returns a 40x response again, it is sent to Basic Authentication handler to Handle. This Handler method will raise a `ValueError` when presented with an authentication scheme other than Digest or Basic.

Modifié dans la version 3.3 : Raise `ValueError` on unsupported Authentication Scheme.

class `urllib.request.`**`ProxyDigestAuthHandler`**(*password_mgr=None*)

Handle authentication with the proxy. *password_mgr*, if given, should be something that is compatible with `HTTPPasswordMgr`; refer to section *HTTPPasswordMgr Objects* for information on the interface that must be supported.

class `urllib.request.`**`HTTPHandler`**

A class to handle opening of HTTP URLs.

class `urllib.request.`**`HTTPSHandler`**(*debuglevel=0*, *context=None*, *check_hostname=None*)

A class to handle opening of HTTPS URLs. *context* and *check_hostname* have the same meaning as in `http.client.HTTPSConnection`.

Modifié dans la version 3.2 : *context* and *check_hostname* were added.

class `urllib.request.`**`FileHandler`**

Open local files.

class `urllib.request.`**`DataHandler`**

Open data URLs.

Ajouté dans la version 3.4.

class `urllib.request.`**`FTPHandler`**

Open FTP URLs.

class `urllib.request.`**`CacheFTPHandler`**

Open FTP URLs, keeping a cache of open FTP connections to minimize delays.

class `urllib.request.`**`UnknownHandler`**

A catch-all class to handle unknown URLs.

class `urllib.request.`**`HTTPErrorProcessor`**

Process HTTP error responses.

22.4.1 Request Objects

The following methods describe `Request`'s public interface, and so all may be overridden in subclasses. It also defines several public attributes that can be used by clients to inspect the parsed request.

`Request.`**`full_url`**

The original URL passed to the constructor.

Modifié dans la version 3.4.

Request.full_url is a property with setter, getter and a deleter. Getting `full_url` returns the original request URL with the fragment, if it was present.

`Request.`**`type`**

The URI scheme.

`Request.`**`host`**

The URI authority, typically a host, but may also contain a port separated by a colon.

`Request.origin_req_host`

The original host for the request, without port.

`Request.selector`

The URI path. If the `Request` uses a proxy, then selector will be the full URL that is passed to the proxy.

`Request.data`

The entity body for the request, or `None` if not specified.

Modifié dans la version 3.4 : Changing value of `Request.data` now deletes "Content-Length" header if it was previously set or calculated.

`Request.unverifiable`

boolean, indicates whether the request is unverifiable as defined by **RFC 2965**.

`Request.method`

The HTTP request method to use. By default its value is `None`, which means that `get_method()` will do its normal computation of the method to be used. Its value can be set (thus overriding the default computation in `get_method()`) either by providing a default value by setting it at the class level in a `Request` subclass, or by passing a value in to the `Request` constructor via the *method* argument.

Ajouté dans la version 3.3.

Modifié dans la version 3.4 : A default value can now be set in subclasses ; previously it could only be set via the constructor argument.

`Request.get_method()`

Return a string indicating the HTTP request method. If `Request.method` is not `None`, return its value, otherwise return `'GET'` if `Request.data` is `None`, or `'POST'` if it's not. This is only meaningful for HTTP requests.

Modifié dans la version 3.3 : get_method now looks at the value of `Request.method`.

`Request.add_header(key, val)`

Add another header to the request. Headers are currently ignored by all handlers except HTTP handlers, where they are added to the list of headers sent to the server. Note that there cannot be more than one header with the same name, and later calls will overwrite previous calls in case the *key* collides. Currently, this is no loss of HTTP functionality, since all headers which have meaning when used more than once have a (header-specific) way of gaining the same functionality using only one header. Note that headers added using this method are also added to redirected requests.

`Request.add_unredirected_header(key, header)`

Add a header that will not be added to a redirected request.

`Request.has_header(header)`

Return whether the instance has the named header (checks both regular and unredirected).

`Request.remove_header(header)`

Remove named header from the request instance (both from regular and unredirected headers).

Ajouté dans la version 3.4.

`Request.get_full_url()`

Return the URL given in the constructor.

Modifié dans la version 3.4.

Returns `Request.full_url`

`Request.set_proxy(host, type)`

Prepare the request by connecting to a proxy server. The *host* and *type* will replace those of the instance, and the instance's selector will be the original URL given in the constructor.

`Request.get_header(header_name, default=None)`

Return the value of the given header. If the header is not present, return the default value.

`Request.header_items()`

Return a list of tuples (header_name, header_value) of the Request headers.

Modifié dans la version 3.4 : The request methods add_data, has_data, get_data, get_type, get_host, get_selector, get_origin_req_host and is_unverifiable that were deprecated since 3.3 have been removed.

22.4.2 OpenerDirector Objects

`OpenerDirector` instances have the following methods :

`OpenerDirector.add_handler(handler)`

handler should be an instance of `BaseHandler`. The following methods are searched, and added to the possible chains (note that HTTP errors are a special case). Note that, in the following, *protocol* should be replaced with the actual protocol to handle, for example `http_response()` would be the HTTP protocol response handler. Also *type* should be replaced with the actual HTTP code, for example `http_error_404()` would handle HTTP 404 errors.

— `<protocol>_open()` --- signal that the handler knows how to open *protocol* URLs.

See `BaseHandler.<protocol>_open()` for more information.

— `http_error_<type>()` --- signal that the handler knows how to handle HTTP errors with HTTP error code *type*.

See `BaseHandler.http_error_<nnn>()` for more information.

— `<protocol>_error()` --- signal that the handler knows how to handle errors from (non-`http`) *protocol*.

— `<protocol>_request()` --- signal that the handler knows how to pre-process *protocol* requests.

See `BaseHandler.<protocol>_request()` for more information.

— `<protocol>_response()` --- signal that the handler knows how to post-process *protocol* responses.

See `BaseHandler.<protocol>_response()` for more information.

`OpenerDirector.open(url, data=None[, timeout])`

Open the given *url* (which can be a request object or a string), optionally passing the given *data*. Arguments, return values and exceptions raised are the same as those of `urlopen()` (which simply calls the `open()` method on the currently installed global `OpenerDirector`). The optional *timeout* parameter specifies a timeout in seconds for blocking operations like the connection attempt (if not specified, the global default timeout setting will be used). The timeout feature actually works only for HTTP, HTTPS and FTP connections.

`OpenerDirector.error(proto, *args)`

Handle an error of the given protocol. This will call the registered error handlers for the given protocol with the given arguments (which are protocol specific). The HTTP protocol is a special case which uses the HTTP response code to determine the specific error handler ; refer to the `http_error_<type>()` methods of the handler classes.

Return values and exceptions raised are the same as those of `urlopen()`.

OpenerDirector objects open URLs in three stages :

The order in which these methods are called within each stage is determined by sorting the handler instances.

1. Every handler with a method named like `<protocol>_request()` has that method called to pre-process the request.
2. Handlers with a method named like `<protocol>_open()` are called to handle the request. This stage ends when a handler either returns a non-`None` value (ie. a response), or raises an exception (usually `URLError`). Exceptions are allowed to propagate.

   In fact, the above algorithm is first tried for methods named `default_open()`. If all such methods return `None`, the algorithm is repeated for methods named like `<protocol>_open()`. If all such methods return `None`, the algorithm is repeated for methods named `unknown_open()`.

   Note that the implementation of these methods may involve calls of the parent `OpenerDirector` instance's `open()` and `error()` methods.

3. Every handler with a method named like `<protocol>_response()` has that method called to post-process the response.

22.4.3 BaseHandler Objects

`BaseHandler` objects provide a couple of methods that are directly useful, and others that are meant to be used by derived classes. These are intended for direct use :

`BaseHandler.add_parent(director)`

Add a director as parent.

`BaseHandler.close()`

Remove any parents.

The following attribute and methods should only be used by classes derived from `BaseHandler`.

> **Note**
>
> The convention has been adopted that subclasses defining `<protocol>_request()` or `<protocol>_response()` methods are named `*Processor`; all others are named `*Handler`.

`BaseHandler.parent`

A valid `OpenerDirector`, which can be used to open using a different protocol, or handle errors.

`BaseHandler.default_open(req)`

This method is *not* defined in `BaseHandler`, but subclasses should define it if they want to catch all URLs.

This method, if implemented, will be called by the parent `OpenerDirector`. It should return a file-like object as described in the return value of the `open()` method of `OpenerDirector`, or `None`. It should raise `URLError`, unless a truly exceptional thing happens (for example, `MemoryError` should not be mapped to `URLError`).

This method will be called before any protocol-specific open method.

`BaseHandler.<protocol>_open(req)`

This method is *not* defined in `BaseHandler`, but subclasses should define it if they want to handle URLs with the given protocol.

This method, if defined, will be called by the parent `OpenerDirector`. Return values should be the same as for `default_open()`.

`BaseHandler.unknown_open(req)`

This method is *not* defined in `BaseHandler`, but subclasses should define it if they want to catch all URLs with no specific registered handler to open it.

This method, if implemented, will be called by the `parent OpenerDirector`. Return values should be the same as for `default_open()`.

`BaseHandler.http_error_default(req, fp, code, msg, hdrs)`

This method is *not* defined in `BaseHandler`, but subclasses should override it if they intend to provide a catch-all for otherwise unhandled HTTP errors. It will be called automatically by the `OpenerDirector` getting the error, and should not normally be called in other circumstances.

req will be a `Request` object, *fp* will be a file-like object with the HTTP error body, *code* will be the three-digit code of the error, *msg* will be the user-visible explanation of the code and *hdrs* will be a mapping object with the headers of the error.

Return values and exceptions raised should be the same as those of `urlopen()`.

`BaseHandler.http_error_<nnn>(req, fp, code, msg, hdrs)`

nnn should be a three-digit HTTP error code. This method is also not defined in `BaseHandler`, but will be called, if it exists, on an instance of a subclass, when an HTTP error with code *nnn* occurs.

Subclasses should override this method to handle specific HTTP errors.

Arguments, return values and exceptions raised should be the same as for `http_error_default()`.

BaseHandler.<protocol>_request(req)

This method is *not* defined in `BaseHandler`, but subclasses should define it if they want to pre-process requests of the given protocol.

This method, if defined, will be called by the parent `OpenerDirector`. *req* will be a `Request` object. The return value should be a `Request` object.

BaseHandler.<protocol>_response(req, response)

This method is *not* defined in `BaseHandler`, but subclasses should define it if they want to post-process responses of the given protocol.

This method, if defined, will be called by the parent `OpenerDirector`. *req* will be a `Request` object. *response* will be an object implementing the same interface as the return value of `urlopen()`. The return value should implement the same interface as the return value of `urlopen()`.

22.4.4 HTTPRedirectHandler Objects

> **Note**
>
> Some HTTP redirections require action from this module's client code. If this is the case, `HTTPError` is raised. See **RFC 2616** for details of the precise meanings of the various redirection codes.
>
> An `HTTPError` exception raised as a security consideration if the HTTPRedirectHandler is presented with a redirected URL which is not an HTTP, HTTPS or FTP URL.

HTTPRedirectHandler.**redirect_request**(*req*, *fp*, *code*, *msg*, *hdrs*, *newurl*)

Return a `Request` or `None` in response to a redirect. This is called by the default implementations of the `http_error_30*()` methods when a redirection is received from the server. If a redirection should take place, return a new `Request` to allow `http_error_30*()` to perform the redirect to *newurl*. Otherwise, raise `HTTPError` if no other handler should try to handle this URL, or return `None` if you can't but another handler might.

> **Note**
>
> The default implementation of this method does not strictly follow **RFC 2616**, which says that 301 and 302 responses to `POST` requests must not be automatically redirected without confirmation by the user. In reality, browsers do allow automatic redirection of these responses, changing the POST to a `GET`, and the default implementation reproduces this behavior.

HTTPRedirectHandler.**http_error_301**(*req*, *fp*, *code*, *msg*, *hdrs*)

Redirect to the `Location:` or `URI:` URL. This method is called by the parent `OpenerDirector` when getting an HTTP 'moved permanently' response.

HTTPRedirectHandler.**http_error_302**(*req*, *fp*, *code*, *msg*, *hdrs*)

The same as `http_error_301()`, but called for the 'found' response.

HTTPRedirectHandler.**http_error_303**(*req*, *fp*, *code*, *msg*, *hdrs*)

The same as `http_error_301()`, but called for the 'see other' response.

HTTPRedirectHandler.**http_error_307**(*req*, *fp*, *code*, *msg*, *hdrs*)

The same as `http_error_301()`, but called for the 'temporary redirect' response. It does not allow changing the request method from `POST` to `GET`.

HTTPRedirectHandler.**http_error_308**(*req*, *fp*, *code*, *msg*, *hdrs*)

The same as `http_error_301()`, but called for the 'permanent redirect' response. It does not allow changing the request method from `POST` to `GET`.

Ajouté dans la version 3.11.

22.4.5 HTTPCookieProcessor Objects

`HTTPCookieProcessor` instances have one attribute :

HTTPCookieProcessor.**cookiejar**

The `http.cookiejar.CookieJar` in which cookies are stored.

22.4.6 ProxyHandler Objects

ProxyHandler.<protocol>_open(request)

The `ProxyHandler` will have a method `<protocol>_open()` for every *protocol* which has a proxy in the *proxies* dictionary given in the constructor. The method will modify requests to go through the proxy, by calling `request.set_proxy()`, and call the next handler in the chain to actually execute the protocol.

22.4.7 HTTPPasswordMgr Objects

These methods are available on `HTTPPasswordMgr` and `HTTPPasswordMgrWithDefaultRealm` objects.

HTTPPasswordMgr.**add_password**(*realm*, *uri*, *user*, *passwd*)

uri can be either a single URI, or a sequence of URIs. *realm*, *user* and *passwd* must be strings. This causes `(user, passwd)` to be used as authentication tokens when authentication for *realm* and a super-URI of any of the given URIs is given.

HTTPPasswordMgr.**find_user_password**(*realm*, *authuri*)

Get user/password for given realm and URI, if any. This method will return `(None, None)` if there is no matching user/password.

For `HTTPPasswordMgrWithDefaultRealm` objects, the realm `None` will be searched if the given *realm* has no matching user/password.

22.4.8 HTTPPasswordMgrWithPriorAuth Objects

This password manager extends `HTTPPasswordMgrWithDefaultRealm` to support tracking URIs for which authentication credentials should always be sent.

HTTPPasswordMgrWithPriorAuth.**add_password**(*realm*, *uri*, *user*, *passwd*, *is_authenticated=False*)

realm, *uri*, *user*, *passwd* are as for `HTTPPasswordMgr.add_password()`. *is_authenticated* sets the initial value of the `is_authenticated` flag for the given URI or list of URIs. If *is_authenticated* is specified as `True`, *realm* is ignored.

HTTPPasswordMgrWithPriorAuth.**find_user_password**(*realm*, *authuri*)

Same as for `HTTPPasswordMgrWithDefaultRealm` objects

HTTPPasswordMgrWithPriorAuth.**update_authenticated**(*self*, *uri*, *is_authenticated=False*)

Update the `is_authenticated` flag for the given *uri* or list of URIs.

HTTPPasswordMgrWithPriorAuth.**is_authenticated**(*self*, *authuri*)

Returns the current state of the `is_authenticated` flag for the given URI.

22.4.9 AbstractBasicAuthHandler Objects

`AbstractBasicAuthHandler.`**`http_error_auth_reqed`**(*authreq*, *host*, *req*, *headers*)

Handle an authentication request by getting a user/password pair, and re-trying the request. *authreq* should be the name of the header where the information about the realm is included in the request, *host* specifies the URL and path to authenticate for, *req* should be the (failed) `Request` object, and *headers* should be the error headers.

host is either an authority (e.g. `"python.org"`) or a URL containing an authority component (e.g. `"http://python.org/"`). In either case, the authority must not contain a userinfo component (so, `"python.org"` and `"python.org:80"` are fine, `"joe:password@python.org"` is not).

22.4.10 HTTPBasicAuthHandler Objects

`HTTPBasicAuthHandler.`**`http_error_401`**(*req*, *fp*, *code*, *msg*, *hdrs*)

Retry the request with authentication information, if available.

22.4.11 ProxyBasicAuthHandler Objects

`ProxyBasicAuthHandler.`**`http_error_407`**(*req*, *fp*, *code*, *msg*, *hdrs*)

Retry the request with authentication information, if available.

22.4.12 AbstractDigestAuthHandler Objects

`AbstractDigestAuthHandler.`**`http_error_auth_reqed`**(*authreq*, *host*, *req*, *headers*)

authreq should be the name of the header where the information about the realm is included in the request, *host* should be the host to authenticate to, *req* should be the (failed) `Request` object, and *headers* should be the error headers.

22.4.13 HTTPDigestAuthHandler Objects

`HTTPDigestAuthHandler.`**`http_error_401`**(*req*, *fp*, *code*, *msg*, *hdrs*)

Retry the request with authentication information, if available.

22.4.14 ProxyDigestAuthHandler Objects

`ProxyDigestAuthHandler.`**`http_error_407`**(*req*, *fp*, *code*, *msg*, *hdrs*)

Retry the request with authentication information, if available.

22.4.15 HTTPHandler Objects

`HTTPHandler.`**`http_open`**(*req*)

Send an HTTP request, which can be either GET or POST, depending on `req.has_data()`.

22.4.16 HTTPSHandler Objects

`HTTPSHandler.`**`https_open`**(*req*)

Send an HTTPS request, which can be either GET or POST, depending on `req.has_data()`.

22.4.17 FileHandler Objects

`FileHandler.`**`file_open`**(*req*)

Open the file locally, if there is no host name, or the host name is `'localhost'`.

Modifié dans la version 3.2 : This method is applicable only for local hostnames. When a remote hostname is given, a `URLError` is raised.

22.4.18 DataHandler Objects

`DataHandler.data_open(req)`

Read a data URL. This kind of URL contains the content encoded in the URL itself. The data URL syntax is specified in RFC 2397. This implementation ignores white spaces in base64 encoded data URLs so the URL may be wrapped in whatever source file it comes from. But even though some browsers don't mind about a missing padding at the end of a base64 encoded data URL, this implementation will raise a `ValueError` in that case.

22.4.19 FTPHandler Objects

`FTPHandler.ftp_open(req)`

Open the FTP file indicated by *req*. The login is always done with empty username and password.

22.4.20 CacheFTPHandler Objects

`CacheFTPHandler` objects are `FTPHandler` objects with the following additional methods :

`CacheFTPHandler.setTimeout(t)`

Set timeout of connections to *t* seconds.

`CacheFTPHandler.setMaxConns(m)`

Set maximum number of cached connections to *m*.

22.4.21 UnknownHandler Objects

`UnknownHandler.unknown_open()`

Raise a `URLError` exception.

22.4.22 HTTPErrorProcessor Objects

`HTTPErrorProcessor.http_response(request, response)`

Process HTTP error responses.

For 200 error codes, the response object is returned immediately.

For non-200 error codes, this simply passes the job on to the `http_error_<type>()` handler methods, via `OpenerDirector.error()`. Eventually, `HTTPDefaultErrorHandler` will raise an `HTTPError` if no other handler handles the error.

`HTTPErrorProcessor.https_response(request, response)`

Process HTTPS error responses.

The behavior is same as `http_response()`.

22.4.23 Exemples

In addition to the examples below, more examples are given in urllib-howto.

This example gets the python.org main page and displays the first 300 bytes of it.

```
>>> import urllib.request
>>> with urllib.request.urlopen('http://www.python.org/') as f:
...     print(f.read(300))
...
b'<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">\n\n\n<html
xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">\n\n<head>\n
<meta http-equiv="content-type" content="text/html; charset=utf-8" />\n
<title>Python Programming '
```

Note that urlopen returns a bytes object. This is because there is no way for urlopen to automatically determine the encoding of the byte stream it receives from the HTTP server. In general, a program will decode the returned bytes object to string once it determines or guesses the appropriate encoding.

The following W3C document, https://www.w3.org/International/O-charset, lists the various ways in which an (X)HTML or an XML document could have specified its encoding information.

As the python.org website uses *utf-8* encoding as specified in its meta tag, we will use the same for decoding the bytes object.

```
>>> with urllib.request.urlopen('http://www.python.org/') as f:
...     print(f.read(100).decode('utf-8'))
...
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtm
```

It is also possible to achieve the same result without using the *context manager* approach.

```
>>> import urllib.request
>>> f = urllib.request.urlopen('http://www.python.org/')
>>> print(f.read(100).decode('utf-8'))
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtm
```

In the following example, we are sending a data-stream to the stdin of a CGI and reading the data it returns to us. Note that this example will only work when the Python installation supports SSL.

```
>>> import urllib.request
>>> req = urllib.request.Request(url='https://localhost/cgi-bin/test.cgi',
...                              data=b'This data is passed to stdin of the CGI')
>>> with urllib.request.urlopen(req) as f:
...     print(f.read().decode('utf-8'))
...
Got Data: "This data is passed to stdin of the CGI"
```

The code for the sample CGI used in the above example is :

```
#!/usr/bin/env python
import sys
data = sys.stdin.read()
print('Content-type: text/plain\n\nGot Data: "%s"' % data)
```

Here is an example of doing a `PUT` request using *`Request`* :

```
import urllib.request
DATA = b'some data'
req = urllib.request.Request(url='http://localhost:8080', data=DATA, method='PUT')
with urllib.request.urlopen(req) as f:
    pass
print(f.status)
print(f.reason)
```

Use of Basic HTTP Authentication :

```
import urllib.request
# Create an OpenerDirector with support for Basic HTTP Authentication...
auth_handler = urllib.request.HTTPBasicAuthHandler()
auth_handler.add_password(realm='PDQ Application',
                          uri='https://mahler:8092/site-updates.py',
                          user='klem',
```

(suite sur la page suivante)

(suite de la page précédente)

```
                            passwd='kadidd!ehopper')
opener = urllib.request.build_opener(auth_handler)
# ...and install it globally so it can be used with urlopen.
urllib.request.install_opener(opener)
urllib.request.urlopen('http://www.example.com/login.html')
```

`build_opener()` provides many handlers by default, including a `ProxyHandler`. By default, `ProxyHandler` uses the environment variables named `<scheme>_proxy`, where `<scheme>` is the URL scheme involved. For example, the `http_proxy` environment variable is read to obtain the HTTP proxy's URL.

This example replaces the default `ProxyHandler` with one that uses programmatically supplied proxy URLs, and adds proxy authorization support with `ProxyBasicAuthHandler`.

```
proxy_handler = urllib.request.ProxyHandler({'http': 'http://www.example.com:3128/
↪'})
proxy_auth_handler = urllib.request.ProxyBasicAuthHandler()
proxy_auth_handler.add_password('realm', 'host', 'username', 'password')

opener = urllib.request.build_opener(proxy_handler, proxy_auth_handler)
# This time, rather than install the OpenerDirector, we use it directly:
opener.open('http://www.example.com/login.html')
```

Adding HTTP headers :

Use the *headers* argument to the `Request` constructor, or :

```
import urllib.request
req = urllib.request.Request('http://www.example.com/')
req.add_header('Referer', 'http://www.python.org/')
# Customize the default User-Agent header value:
req.add_header('User-Agent', 'urllib-example/0.1 (Contact: . . .)')
r = urllib.request.urlopen(req)
```

`OpenerDirector` automatically adds a `User-Agent` header to every `Request`. To change this :

```
import urllib.request
opener = urllib.request.build_opener()
opener.addheaders = [('User-agent', 'Mozilla/5.0')]
opener.open('http://www.example.com/')
```

Also, remember that a few standard headers (`Content-Length`, `Content-Type` and `Host`) are added when the `Request` is passed to `urlopen()` (or `OpenerDirector.open()`).

Here is an example session that uses the `GET` method to retrieve a URL containing parameters :

```
>>> import urllib.request
>>> import urllib.parse
>>> params = urllib.parse.urlencode({'spam': 1, 'eggs': 2, 'bacon': 0})
>>> url = "http://www.musi-cal.com/cgi-bin/query?%s" % params
>>> with urllib.request.urlopen(url) as f:
...     print(f.read().decode('utf-8'))
...
```

The following example uses the `POST` method instead. Note that params output from urlencode is encoded to bytes before it is sent to urlopen as data :

```
>>> import urllib.request
>>> import urllib.parse
>>> data = urllib.parse.urlencode({'spam': 1, 'eggs': 2, 'bacon': 0})
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> data = data.encode('ascii')
>>> with urllib.request.urlopen("http://requestb.in/xrbl82xr", data) as f:
...     print(f.read().decode('utf-8'))
...
```

The following example uses an explicitly specified HTTP proxy, overriding environment settings :

```
>>> import urllib.request
>>> proxies = {'http': 'http://proxy.example.com:8080/'}
>>> opener = urllib.request.build_opener(urllib.request.ProxyHandler(proxies))
>>> with opener.open("http://www.python.org") as f:
...     f.read().decode('utf-8')
...
```

The following example uses no proxies at all, overriding environment settings :

```
>>> import urllib.request
>>> opener = urllib.request.build_opener(urllib.request.ProxyHandler({}))
>>> with opener.open("http://www.python.org/") as f:
...     f.read().decode('utf-8')
...
```

22.4.24 Legacy interface

The following functions and classes are ported from the Python 2 module `urllib` (as opposed to `urllib2`). They might become deprecated at some point in the future.

`urllib.request.`**`urlretrieve`**(*url*, *filename=None*, *reporthook=None*, *data=None*)

Copy a network object denoted by a URL to a local file. If the URL points to a local file, the object will not be copied unless filename is supplied. Return a tuple `(filename, headers)` where *filename* is the local file name under which the object can be found, and *headers* is whatever the `info()` method of the object returned by `urlopen()` returned (for a remote object). Exceptions are the same as for `urlopen()`.

The second argument, if present, specifies the file location to copy to (if absent, the location will be a tempfile with a generated name). The third argument, if present, is a callable that will be called once on establishment of the network connection and once after each block read thereafter. The callable will be passed three arguments ; a count of blocks transferred so far, a block size in bytes, and the total size of the file. The third argument may be `-1` on older FTP servers which do not return a file size in response to a retrieval request.

The following example illustrates the most common usage scenario :

```
>>> import urllib.request
>>> local_filename, headers = urllib.request.urlretrieve('http://python.org/')
>>> html = open(local_filename)
>>> html.close()
```

If the *url* uses the `http:` scheme identifier, the optional *data* argument may be given to specify a `POST` request (normally the request type is `GET`). The *data* argument must be a bytes object in standard `application/x-www-form-urlencoded` format ; see the `urllib.parse.urlencode()` function.

`urlretrieve()` will raise `ContentTooShortError` when it detects that the amount of data available was less than the expected amount (which is the size reported by a *Content-Length* header). This can occur, for example, when the download is interrupted.

The *Content-Length* is treated as a lower bound : if there's more data to read, urlretrieve reads more data, but if less data is available, it raises the exception.

You can still retrieve the downloaded data in this case, it is stored in the `content` attribute of the exception instance.

If no *Content-Length* header was supplied, urlretrieve can not check the size of the data it has downloaded, and just returns it. In this case you just have to assume that the download was successful.

`urllib.request.urlcleanup()`

Cleans up temporary files that may have been left behind by previous calls to `urlretrieve()`.

22.4.25 `urllib.request` Restrictions

— Currently, only the following protocols are supported : HTTP (versions 0.9 and 1.0), FTP, local files, and data URLs.

  Modifié dans la version 3.4 : Added support for data URLs.

— The caching feature of `urlretrieve()` has been disabled until someone finds the time to hack proper processing of Expiration time headers.

— There should be a function to query whether a particular URL is in the cache.

— For backward compatibility, if a URL appears to point to a local file but the file can't be opened, the URL is re-interpreted using the FTP protocol. This can sometimes cause confusing error messages.

— The `urlopen()` and `urlretrieve()` functions can cause arbitrarily long delays while waiting for a network connection to be set up. This means that it is difficult to build an interactive web client using these functions without using threads.

— The data returned by `urlopen()` or `urlretrieve()` is the raw data returned by the server. This may be binary data (such as an image), plain text or (for example) HTML. The HTTP protocol provides type information in the reply header, which can be inspected by looking at the `Content-Type` header. If the returned data is HTML, you can use the module `html.parser` to parse it.

— The code handling the FTP protocol cannot differentiate between a file and a directory. This can lead to unexpected behavior when attempting to read a URL that points to a file that is not accessible. If the URL ends in a `/`, it is assumed to refer to a directory and will be handled accordingly. But if an attempt to read a file leads to a 550 error (meaning the URL cannot be found or is not accessible, often for permission reasons), then the path is treated as a directory in order to handle the case when a directory is specified by a URL but the trailing `/` has been left off. This can cause misleading results when you try to fetch a file whose read permissions make it inaccessible ; the FTP code will try to read it, fail with a 550 error, and then perform a directory listing for the unreadable file. If fine-grained control is needed, consider using the `ftplib` module.

22.5 `urllib.response` --- Response classes used by urllib

The `urllib.response` module defines functions and classes which define a minimal file-like interface, including `read()` and `readline()`. Functions defined by this module are used internally by the `urllib.request` module. The typical response object is a `urllib.response.addinfourl` instance :

class `urllib.response.addinfourl`

url

L'URL de la ressource récupérée, utilisée habituellement pour déterminer si une redirection a été suivie.

headers

Returns the headers of the response in the form of an `EmailMessage` instance.

status

Ajouté dans la version 3.9.

Code d'état renvoyé par le serveur.

geturl()

Obsolète depuis la version 3.9 : Deprecated in favor of `url`.

info()

Obsolète depuis la version 3.9 : Deprecated in favor of `headers`.

code

Obsolète depuis la version 3.9 : Deprecated in favor of `status`.

getcode()

Obsolète depuis la version 3.9 : Deprecated in favor of *status*.

22.6 urllib.parse --- Parse URLs into components

Code source : Lib/urllib/parse.py

This module defines a standard interface to break Uniform Resource Locator (URL) strings up in components (addressing scheme, network location, path etc.), to combine the components back into a URL string, and to convert a "relative URL" to an absolute URL given a "base URL."

The module has been designed to match the internet RFC on Relative Uniform Resource Locators. It supports the following URL schemes : `file`, `ftp`, `gopher`, `hdl`, `http`, `https`, `imap`, `itms-services`, `mailto`, `mms`, `news`, `nntp`, `prospero`, `rsync`, `rtsp`, `rtsps`, `rtspu`, `sftp`, `shttp`, `sip`, `sips`, `snews`, `svn`, `svn+ssh`, `telnet`, `wais`, `ws`, `wss`.

Particularité de l'implémentation CPython : The inclusion of the `itms-services` URL scheme can prevent an app from passing Apple's App Store review process for the macOS and iOS App Stores. Handling for the `itms-services` scheme is always removed on iOS ; on macOS, it *may* be removed if CPython has been built with the `--with-app-store-compliance` option.

The *urllib.parse* module defines functions that fall into two broad categories : URL parsing and URL quoting. These are covered in detail in the following sections.

This module's functions use the deprecated term `netloc` (or `net_loc`), which was introduced in **RFC 1808**. However, this term has been obsoleted by **RFC 3986**, which introduced the term `authority` as its replacement. The use of `netloc` is continued for backward compatibility.

22.6.1 URL Parsing

The URL parsing functions focus on splitting a URL string into its components, or on combining URL components into a URL string.

`urllib.parse.`**urlparse**(*urlstring*, *scheme=''*, *allow_fragments=True*)

Parse a URL into six components, returning a 6-item *named tuple*. This corresponds to the general structure of a URL : `scheme://netloc/path;parameters?query#fragment`. Each tuple item is a string, possibly empty. The components are not broken up into smaller parts (for example, the network location is a single string), and % escapes are not expanded. The delimiters as shown above are not part of the result, except for a leading slash in the *path* component, which is retained if present. For example :

```
>>> from urllib.parse import urlparse
>>> urlparse("scheme://netloc/path;parameters?query#fragment")
ParseResult(scheme='scheme', netloc='netloc', path='/path;parameters', params='
↪',
            query='query', fragment='fragment')
>>> o = urlparse("http://docs.python.org:80/3/library/urllib.parse.html?"
...              "highlight=params#url-parsing")
>>> o
ParseResult(scheme='http', netloc='docs.python.org:80',
            path='/3/library/urllib.parse.html', params='',
            query='highlight=params', fragment='url-parsing')
>>> o.scheme
'http'
>>> o.netloc
'docs.python.org:80'
>>> o.hostname
'docs.python.org'
>>> o.port
```

(suite sur la page suivante)

(suite de la page précédente)

```
80
>>> o._replace(fragment="").geturl()
'http://docs.python.org:80/3/library/urllib.parse.html?highlight=params'
```

Following the syntax specifications in **RFC 1808**, urlparse recognizes a netloc only if it is properly introduced by '//'. Otherwise the input is presumed to be a relative URL and thus to start with a path component.

```
>>> from urllib.parse import urlparse
>>> urlparse('//www.cwi.nl:80/%7Eguido/Python.html')
ParseResult(scheme='', netloc='www.cwi.nl:80', path='/%7Eguido/Python.html',
            params='', query='', fragment='')
>>> urlparse('www.cwi.nl/%7Eguido/Python.html')
ParseResult(scheme='', netloc='', path='www.cwi.nl/%7Eguido/Python.html',
            params='', query='', fragment='')
>>> urlparse('help/Python.html')
ParseResult(scheme='', netloc='', path='help/Python.html', params='',
            query='', fragment='')
```

The *scheme* argument gives the default addressing scheme, to be used only if the URL does not specify one. It should be the same type (text or bytes) as *urlstring*, except that the default value `''` is always allowed, and is automatically converted to `b''` if appropriate.

If the *allow_fragments* argument is false, fragment identifiers are not recognized. Instead, they are parsed as part of the path, parameters or query component, and `fragment` is set to the empty string in the return value.

The return value is a *named tuple*, which means that its items can be accessed by index or as named attributes, which are :

Attribut	Index	Valeur	Value if not present
`scheme`	0	URL scheme specifier	*scheme* parameter
`netloc`	1	Network location part	empty string
`path`	2	Hierarchical path	empty string
`params`	3	Parameters for last path element	empty string
`query`	4	Query component	empty string
`fragment`	5	Fragment identifier	empty string
`username`		User name	*None*
`password`		Password	*None*
`hostname`		Host name (lower case)	*None*
`port`		Port number as integer, if present	*None*

Reading the `port` attribute will raise a *ValueError* if an invalid port is specified in the URL. See section *Structured Parse Results* for more information on the result object.

Unmatched square brackets in the `netloc` attribute will raise a *ValueError*.

Characters in the `netloc` attribute that decompose under NFKC normalization (as used by the IDNA encoding) into any of `/`, `?`, `#`, `@`, or `:` will raise a *ValueError*. If the URL is decomposed before parsing, no error will be raised.

As is the case with all named tuples, the subclass has a few additional methods and attributes that are particularly useful. One such method is `_replace()`. The `_replace()` method will return a new ParseResult object replacing specified fields with new values.

```
>>> from urllib.parse import urlparse
>>> u = urlparse('//www.cwi.nl:80/%7Eguido/Python.html')
>>> u
ParseResult(scheme='', netloc='www.cwi.nl:80', path='/%7Eguido/Python.html',
            params='', query='', fragment='')
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> u._replace(scheme='http')
ParseResult(scheme='http', netloc='www.cwi.nl:80', path='/%7Eguido/Python.html
↪',
            params='', query='', fragment='')
```

Avertissement

`urlparse()` does not perform validation. See *URL parsing security* for details.

Modifié dans la version 3.2 : Added IPv6 URL parsing capabilities.

Modifié dans la version 3.3 : The fragment is now parsed for all URL schemes (unless *allow_fragments* is false), in accordance with **RFC 3986**. Previously, an allowlist of schemes that support fragments existed.

Modifié dans la version 3.6 : Out-of-range port numbers now raise `ValueError`, instead of returning `None`.

Modifié dans la version 3.8 : Characters that affect netloc parsing under NFKC normalization will now raise `ValueError`.

`urllib.parse.`**`parse_qs`**(*qs*, *keep_blank_values=False*, *strict_parsing=False*, *encoding='utf-8'*, *errors='replace'*, *max_num_fields=None*, *separator='&'*)

Parse a query string given as a string argument (data of type `application/x-www-form-urlencoded`). Data are returned as a dictionary. The dictionary keys are the unique query variable names and the values are lists of values for each name.

The optional argument *keep_blank_values* is a flag indicating whether blank values in percent-encoded queries should be treated as blank strings. A true value indicates that blanks should be retained as blank strings. The default false value indicates that blank values are to be ignored and treated as if they were not included.

The optional argument *strict_parsing* is a flag indicating what to do with parsing errors. If false (the default), errors are silently ignored. If true, errors raise a `ValueError` exception.

The optional *encoding* and *errors* parameters specify how to decode percent-encoded sequences into Unicode characters, as accepted by the `bytes.decode()` method.

The optional argument *max_num_fields* is the maximum number of fields to read. If set, then throws a `ValueError` if there are more than *max_num_fields* fields read.

The optional argument *separator* is the symbol to use for separating the query arguments. It defaults to `&`.

Use the `urllib.parse.urlencode()` function (with the `doseq` parameter set to `True`) to convert such dictionaries into query strings.

Modifié dans la version 3.2 : Add *encoding* and *errors* parameters.

Modifié dans la version 3.8 : Added *max_num_fields* parameter.

Modifié dans la version 3.10 : Added *separator* parameter with the default value of `&`. Python versions earlier than Python 3.10 allowed using both `;` and `&` as query parameter separator. This has been changed to allow only a single separator key, with `&` as the default separator.

Obsolète depuis la version 3.14 : Accepting objects with false values (like `0` and `[]`) except empty strings and byte-like objects and `None` is now deprecated.

`urllib.parse.`**`parse_qsl`**(*qs*, *keep_blank_values=False*, *strict_parsing=False*, *encoding='utf-8'*, *errors='replace'*, *max_num_fields=None*, *separator='&'*)

Parse a query string given as a string argument (data of type `application/x-www-form-urlencoded`). Data are returned as a list of name, value pairs.

The optional argument *keep_blank_values* is a flag indicating whether blank values in percent-encoded queries should be treated as blank strings. A true value indicates that blanks should be retained as blank strings. The default false value indicates that blank values are to be ignored and treated as if they were not included.

The optional argument *strict_parsing* is a flag indicating what to do with parsing errors. If false (the default), errors are silently ignored. If true, errors raise a `ValueError` exception.

The optional *encoding* and *errors* parameters specify how to decode percent-encoded sequences into Unicode characters, as accepted by the `bytes.decode()` method.

The optional argument *max_num_fields* is the maximum number of fields to read. If set, then throws a `ValueError` if there are more than *max_num_fields* fields read.

The optional argument *separator* is the symbol to use for separating the query arguments. It defaults to `&`.

Use the `urllib.parse.urlencode()` function to convert such lists of pairs into query strings.

Modifié dans la version 3.2 : Add *encoding* and *errors* parameters.

Modifié dans la version 3.8 : Added *max_num_fields* parameter.

Modifié dans la version 3.10 : Added *separator* parameter with the default value of `&`. Python versions earlier than Python 3.10 allowed using both `;` and `&` as query parameter separator. This has been changed to allow only a single separator key, with `&` as the default separator.

`urllib.parse.`**`urlunparse`**(*parts*)

Construct a URL from a tuple as returned by `urlparse()`. The *parts* argument can be any six-item iterable. This may result in a slightly different, but equivalent URL, if the URL that was parsed originally had unnecessary delimiters (for example, a `?` with an empty query; the RFC states that these are equivalent).

`urllib.parse.`**`urlsplit`**(*urlstring*, *scheme=''*, *allow_fragments=True*)

This is similar to `urlparse()`, but does not split the params from the URL. This should generally be used instead of `urlparse()` if the more recent URL syntax allowing parameters to be applied to each segment of the *path* portion of the URL (see **RFC 2396**) is wanted. A separate function is needed to separate the path segments and parameters. This function returns a 5-item *named tuple* :

```
(addressing scheme, network location, path, query, fragment identifier).
```

The return value is a *named tuple*, its items can be accessed by index or as named attributes :

Attribut	Index	Valeur	Value if not present
`scheme`	0	URL scheme specifier	*scheme* parameter
`netloc`	1	Network location part	empty string
`path`	2	Hierarchical path	empty string
`query`	3	Query component	empty string
`fragment`	4	Fragment identifier	empty string
`username`		User name	`None`
`password`		Password	`None`
`hostname`		Host name (lower case)	`None`
`port`		Port number as integer, if present	`None`

Reading the `port` attribute will raise a `ValueError` if an invalid port is specified in the URL. See section *Structured Parse Results* for more information on the result object.

Unmatched square brackets in the `netloc` attribute will raise a `ValueError`.

Characters in the `netloc` attribute that decompose under NFKC normalization (as used by the IDNA encoding) into any of `/`, `?`, `#`, `@`, or `:` will raise a `ValueError`. If the URL is decomposed before parsing, no error will be raised.

Following some of the WHATWG spec that updates RFC 3986, leading C0 control and space characters are stripped from the URL. `\n`, `\r` and tab `\t` characters are removed from the URL at any position.

Avertissement

> `urlsplit()` does not perform validation. See *URL parsing security* for details.

Modifié dans la version 3.6 : Out-of-range port numbers now raise `ValueError`, instead of returning `None`.

Modifié dans la version 3.8 : Characters that affect netloc parsing under NFKC normalization will now raise `ValueError`.

Modifié dans la version 3.10 : ASCII newline and tab characters are stripped from the URL.

Modifié dans la version 3.12 : Leading WHATWG C0 control and space characters are stripped from the URL.

`urllib.parse.`**`urlunsplit`**(*parts*)

Combine the elements of a tuple as returned by `urlsplit()` into a complete URL as a string. The *parts* argument can be any five-item iterable. This may result in a slightly different, but equivalent URL, if the URL that was parsed originally had unnecessary delimiters (for example, a ? with an empty query ; the RFC states that these are equivalent).

`urllib.parse.`**`urljoin`**(*base*, *url*, *allow_fragments=True*)

Construct a full ("absolute") URL by combining a "base URL" (*base*) with another URL (*url*). Informally, this uses components of the base URL, in particular the addressing scheme, the network location and (part of) the path, to provide missing components in the relative URL. For example :

```
>>> from urllib.parse import urljoin
>>> urljoin('http://www.cwi.nl/%7Eguido/Python.html', 'FAQ.html')
'http://www.cwi.nl/%7Eguido/FAQ.html'
```

The *allow_fragments* argument has the same meaning and default as for `urlparse()`.

> **Note**
>
> If *url* is an absolute URL (that is, it starts with `//` or `scheme://`), the *url*'s hostname and/or scheme will be present in the result. For example :
>
> ```
> >>> urljoin('http://www.cwi.nl/%7Eguido/Python.html',
> ...         '//www.python.org/%7Eguido')
> 'http://www.python.org/%7Eguido'
> ```
>
> If you do not want that behavior, preprocess the *url* with `urlsplit()` and `urlunsplit()`, removing possible *scheme* and *netloc* parts.

> **Avertissement**
>
> Because an absolute URL may be passed as the `url` parameter, it is generally **not secure** to use `urljoin` with an attacker-controlled `url`. For example in, `urljoin("https://website.com/users/", username)`, if `username` can contain an absolute URL, the result of `urljoin` will be the absolute URL.

Modifié dans la version 3.5 : Behavior updated to match the semantics defined in **RFC 3986**.

`urllib.parse.`**`urldefrag`**(*url*)

If *url* contains a fragment identifier, return a modified version of *url* with no fragment identifier, and the fragment identifier as a separate string. If there is no fragment identifier in *url*, return *url* unmodified and an empty string.

The return value is a *named tuple*, its items can be accessed by index or as named attributes :

Attribut	Index	Valeur	Value if not present
`url`	0	URL with no fragment	empty string
`fragment`	1	Fragment identifier	empty string

See section *Structured Parse Results* for more information on the result object.

Modifié dans la version 3.2 : Result is a structured object rather than a simple 2-tuple.

`urllib.parse.`**`unwrap`**(*url*)

Extract the url from a wrapped URL (that is, a string formatted as `<URL:scheme://host/path>`, `<scheme://host/path>`, `URL:scheme://host/path` or `scheme://host/path`). If *url* is not a wrapped URL, it is returned without changes.

22.6.2 URL parsing security

The `urlsplit()` and `urlparse()` APIs do not perform **validation** of inputs. They may not raise errors on inputs that other applications consider invalid. They may also succeed on some inputs that might not be considered URLs elsewhere. Their purpose is for practical functionality rather than purity.

Instead of raising an exception on unusual input, they may instead return some component parts as empty strings. Or components may contain more than perhaps they should.

We recommend that users of these APIs where the values may be used anywhere with security implications code defensively. Do some verification within your code before trusting a returned component part. Does that `scheme` make sense ? Is that a sensible `path` ? Is there anything strange about that `hostname` ? etc.

What constitutes a URL is not universally well defined. Different applications have different needs and desired constraints. For instance the living WHATWG spec describes what user facing web clients such as a web browser require. While **RFC 3986** is more general. These functions incorporate some aspects of both, but cannot be claimed compliant with either. The APIs and existing user code with expectations on specific behaviors predate both standards leading us to be very cautious about making API behavior changes.

22.6.3 Parsing ASCII Encoded Bytes

The URL parsing functions were originally designed to operate on character strings only. In practice, it is useful to be able to manipulate properly quoted and encoded URLs as sequences of ASCII bytes. Accordingly, the URL parsing functions in this module all operate on `bytes` and `bytearray` objects in addition to `str` objects.

If `str` data is passed in, the result will also contain only `str` data. If `bytes` or `bytearray` data is passed in, the result will contain only `bytes` data.

Attempting to mix `str` data with `bytes` or `bytearray` in a single function call will result in a `TypeError` being raised, while attempting to pass in non-ASCII byte values will trigger `UnicodeDecodeError`.

To support easier conversion of result objects between `str` and `bytes`, all return values from URL parsing functions provide either an `encode()` method (when the result contains `str` data) or a `decode()` method (when the result contains `bytes` data). The signatures of these methods match those of the corresponding `str` and `bytes` methods (except that the default encoding is `'ascii'` rather than `'utf-8'`). Each produces a value of a corresponding type that contains either `bytes` data (for `encode()` methods) or `str` data (for `decode()` methods).

Applications that need to operate on potentially improperly quoted URLs that may contain non-ASCII data will need to do their own decoding from bytes to characters before invoking the URL parsing methods.

The behaviour described in this section applies only to the URL parsing functions. The URL quoting functions use their own rules when producing or consuming byte sequences as detailed in the documentation of the individual URL quoting functions.

Modifié dans la version 3.2 : URL parsing functions now accept ASCII encoded byte sequences

22.6.4 Structured Parse Results

The result objects from the `urlparse()`, `urlsplit()` and `urldefrag()` functions are subclasses of the `tuple` type. These subclasses add the attributes listed in the documentation for those functions, the encoding and decoding support described in the previous section, as well as an additional method :

`urllib.parse.SplitResult.geturl()`

Return the re-combined version of the original URL as a string. This may differ from the original URL in that the scheme may be normalized to lower case and empty components may be dropped. Specifically, empty parameters, queries, and fragment identifiers will be removed.

For `urldefrag()` results, only empty fragment identifiers will be removed. For `urlsplit()` and `urlparse()` results, all noted changes will be made to the URL returned by this method.

The result of this method remains unchanged if passed back through the original parsing function :

```
>>> from urllib.parse import urlsplit
>>> url = 'HTTP://www.Python.org/doc/#'
>>> r1 = urlsplit(url)
>>> r1.geturl()
'http://www.Python.org/doc/'
>>> r2 = urlsplit(r1.geturl())
>>> r2.geturl()
'http://www.Python.org/doc/'
```

The following classes provide the implementations of the structured parse results when operating on `str` objects :

class `urllib.parse.DefragResult`(*url*, *fragment*)

Concrete class for `urldefrag()` results containing `str` data. The `encode()` method returns a `DefragResultBytes` instance.

Ajouté dans la version 3.2.

class `urllib.parse.ParseResult`(*scheme*, *netloc*, *path*, *params*, *query*, *fragment*)

Concrete class for `urlparse()` results containing `str` data. The `encode()` method returns a `ParseResultBytes` instance.

class `urllib.parse.SplitResult`(*scheme*, *netloc*, *path*, *query*, *fragment*)

Concrete class for `urlsplit()` results containing `str` data. The `encode()` method returns a `SplitResultBytes` instance.

The following classes provide the implementations of the parse results when operating on `bytes` or `bytearray` objects :

class `urllib.parse.DefragResultBytes`(*url*, *fragment*)

Concrete class for `urldefrag()` results containing `bytes` data. The `decode()` method returns a `DefragResult` instance.

Ajouté dans la version 3.2.

class `urllib.parse.ParseResultBytes`(*scheme*, *netloc*, *path*, *params*, *query*, *fragment*)

Concrete class for `urlparse()` results containing `bytes` data. The `decode()` method returns a `ParseResult` instance.

Ajouté dans la version 3.2.

class `urllib.parse.SplitResultBytes`(*scheme*, *netloc*, *path*, *query*, *fragment*)

Concrete class for `urlsplit()` results containing `bytes` data. The `decode()` method returns a `SplitResult` instance.

Ajouté dans la version 3.2.

22.6.5 URL Quoting

The URL quoting functions focus on taking program data and making it safe for use as URL components by quoting special characters and appropriately encoding non-ASCII text. They also support reversing these operations to recreate the original data from the contents of a URL component if that task isn't already covered by the URL parsing functions above.

`urllib.parse.quote`(*string*, *safe='/'*, *encoding=None*, *errors=None*)

Replace special characters in *string* using the `%xx` escape. Letters, digits, and the characters `'_.-~'` are never quoted. By default, this function is intended for quoting the path section of a URL. The optional *safe* parameter specifies additional ASCII characters that should not be quoted --- its default value is `'/'`.

string may be either a `str` or a `bytes` object.

Modifié dans la version 3.7 : Moved from **RFC 2396** to **RFC 3986** for quoting URL strings. "~" is now included in the set of unreserved characters.

The optional *encoding* and *errors* parameters specify how to deal with non-ASCII characters, as accepted by the `str.encode()` method. *encoding* defaults to `'utf-8'`. *errors* defaults to `'strict'`, meaning unsupported characters raise a `UnicodeEncodeError`. *encoding* and *errors* must not be supplied if *string* is a `bytes`, or a `TypeError` is raised.

Note that `quote(string, safe, encoding, errors)` is equivalent to `quote_from_bytes(string.encode(encoding, errors), safe)`.

Example : `quote('/El Niño/')` yields `'/El%20Ni%C3%B1o/'`.

`urllib.parse.quote_plus`(*string*, *safe=''*, *encoding=None*, *errors=None*)

Like `quote()`, but also replace spaces with plus signs, as required for quoting HTML form values when building up a query string to go into a URL. Plus signs in the original string are escaped unless they are included in *safe*. It also does not have *safe* default to `'/'`.

Example : `quote_plus('/El Niño/')` yields `'%2FEl+Ni%C3%B1o%2F'`.

`urllib.parse.quote_from_bytes`(*bytes*, *safe='/'*)

Like `quote()`, but accepts a `bytes` object rather than a `str`, and does not perform string-to-bytes encoding.

Example : `quote_from_bytes(b'a&\xef')` yields `'a%26%EF'`.

`urllib.parse.unquote`(*string*, *encoding='utf-8'*, *errors='replace'*)

Replace `%xx` escapes with their single-character equivalent. The optional *encoding* and *errors* parameters specify how to decode percent-encoded sequences into Unicode characters, as accepted by the `bytes.decode()` method.

string may be either a `str` or a `bytes` object.

encoding defaults to `'utf-8'`. *errors* defaults to `'replace'`, meaning invalid sequences are replaced by a placeholder character.

Example : `unquote('/El%20Ni%C3%B1o/')` yields `'/El Niño/'`.

Modifié dans la version 3.9 : *string* parameter supports bytes and str objects (previously only str).

`urllib.parse.unquote_plus`(*string*, *encoding='utf-8'*, *errors='replace'*)

Like `unquote()`, but also replace plus signs with spaces, as required for unquoting HTML form values.

string must be a `str`.

Example : `unquote_plus('/El+Ni%C3%B1o/')` yields `'/El Niño/'`.

`urllib.parse.unquote_to_bytes`(*string*)

Replace `%xx` escapes with their single-octet equivalent, and return a `bytes` object.

string may be either a `str` or a `bytes` object.

If it is a `str`, unescaped non-ASCII characters in *string* are encoded into UTF-8 bytes.

Example : `unquote_to_bytes('a%26%EF')` yields `b'a&\xef'`.

`urllib.parse.urlencode`(*query*, *doseq=False*, *safe=''*, *encoding=None*, *errors=None*, *quote_via=quote_plus*)

Convert a mapping object or a sequence of two-element tuples, which may contain `str` or `bytes` objects, to a percent-encoded ASCII text string. If the resultant string is to be used as a *data* for POST operation with the `urlopen()` function, then it should be encoded to bytes, otherwise it would result in a `TypeError`.

The resulting string is a series of `key=value` pairs separated by `'&'` characters, where both *key* and *value* are quoted using the *quote_via* function. By default, `quote_plus()` is used to quote the values, which means spaces are quoted as a `'+'` character and '/' characters are encoded as `%2F`, which follows the standard for GET requests (`application/x-www-form-urlencoded`). An alternate function that can be passed as *quote_via* is `quote()`, which will encode spaces as `%20` and not encode '/' characters. For maximum control of what is quoted, use `quote` and specify a value for *safe*.

When a sequence of two-element tuples is used as the *query* argument, the first element of each tuple is a key and the second is a value. The value element in itself can be a sequence and in that case, if the optional parameter *doseq* evaluates to `True`, individual `key=value` pairs separated by `'&'` are generated for each element of the value sequence for the key. The order of parameters in the encoded string will match the order of parameter tuples in the sequence.

The *safe*, *encoding*, and *errors* parameters are passed down to *quote_via* (the *encoding* and *errors* parameters are only passed when a query element is a `str`).

To reverse this encoding process, `parse_qs()` and `parse_qsl()` are provided in this module to parse query strings into Python data structures.

Refer to *urllib examples* to find out how the `urllib.parse.urlencode()` method can be used for generating the query string of a URL or data for a POST request.

Modifié dans la version 3.2 : *query* supports bytes and string objects.

Modifié dans la version 3.5 : Added the *quote_via* parameter.

Obsolète depuis la version 3.14 : Accepting objects with false values (like `0` and `[]`) except empty strings and byte-like objects and `None` is now deprecated.

Voir aussi

WHATWG - URL Living standard
Working Group for the URL Standard that defines URLs, domains, IP addresses, the application/x-www-form-urlencoded format, and their API.

RFC 3986 - Uniform Resource Identifiers
This is the current standard (STD66). Any changes to urllib.parse module should conform to this. Certain deviations could be observed, which are mostly for backward compatibility purposes and for certain de-facto parsing requirements as commonly observed in major browsers.

RFC 2732 - Format for Literal IPv6 Addresses in URL's.
This specifies the parsing requirements of IPv6 URLs.

RFC 2396 - Uniform Resource Identifiers (URI) : Generic Syntax
Document describing the generic syntactic requirements for both Uniform Resource Names (URNs) and Uniform Resource Locators (URLs).

RFC 2368 - The mailto URL scheme.
Parsing requirements for mailto URL schemes.

RFC 1808 - Relative Uniform Resource Locators
This Request For Comments includes the rules for joining an absolute and a relative URL, including a fair number of "Abnormal Examples" which govern the treatment of border cases.

RFC 1738 - Uniform Resource Locators (URL)
This specifies the formal syntax and semantics of absolute URLs.

22.7 `urllib.error` --- Exception classes raised by urllib.request

Code source : Lib/urllib/error.py

Le module `urllib.error` définit les classes des exceptions levées par `urllib.request`. La classe de base de ces exceptions est `URLError`.

Les exceptions suivantes sont levées par `urllib.error` aux cas appropriés :

exception `urllib.error.`**`URLError`**

Les gestionnaires lèvent cette exception (ou des exceptions dérivées) quand ils rencontrent un problème. Elle est une sous-classe de `OSError`.

reason

La raison de cette erreur. Il peut s'agir d'un message textuel ou d'une autre instance d'exception.

Modifié dans la version 3.3 : `URLError` used to be a subtype of `IOError`, which is now an alias of `OSError`.

exception `urllib.error.`**`HTTPError`**(*url*, *code*, *msg*, *hdrs*, *fp*)

Bien qu'étant une exception (une sous-classe de `URLError`), une `HTTPError` peut aussi fonctionner comme une valeur de retour normale et fichier-compatible (la même chose que renvoyé par `urlopen()`). Cela est utile pour gérer les erreurs HTTP exotiques, comme les requêtes d'authentification.

url

Contains the request URL. An alias for *filename* attribute.

code

Un statut HTTP comme défini dans la **RFC 2616**. Cette valeur numérique correspond à une valeur trouvée dans le dictionnaire des codes comme dans `http.server.BaseHTTPRequestHandler.responses`.

reason

This is usually a string explaining the reason for this error. An alias for *msg* attribute.

headers

The HTTP response headers for the HTTP request that caused the `HTTPError`. An alias for *hdrs* attribute.

Ajouté dans la version 3.4.

fp

A file-like object where the HTTP error body can be read from.

exception `urllib.error.`**`ContentTooShortError`**(*msg*, *content*)

This exception is raised when the `urlretrieve()` function detects that the amount of the downloaded data is less than the expected amount (given by the *Content-Length* header).

content

The downloaded (and supposedly truncated) data.

22.8 `urllib.robotparser` --- Parser for robots.txt

Code source : Lib/urllib/robotparser.py

This module provides a single class, `RobotFileParser`, which answers questions about whether or not a particular user agent can fetch a URL on the web site that published the `robots.txt` file. For more details on the structure of `robots.txt` files, see http://www.robotstxt.org/orig.html.

class `urllib.robotparser.`**`RobotFileParser`**(*url=''*)

Cette classe fournit des méthodes pour lire, analyser et répondre aux questions à propos du fichier `robots.txt` disponible à l'adresse *url*.

`set_url`(*url*)

Modifie l'URL référençant le fichier `robots.txt`.

`read`()

Lit le fichier `robots.txt` depuis son URL et envoie le contenu à l'analyseur.

`parse`(*lines*)

Analyse les lignes données en argument.

`can_fetch`(*useragent*, *url*)

Renvoie `True` si *useragent* est autorisé à accéder à *url* selon les règles contenues dans le fichier `robots.txt` analysé.

`mtime`()

Renvoie le temps auquel le fichier `robots.txt` a été téléchargé pour la dernière fois. Cela est utile pour des *web spiders* de longue durée qui doivent vérifier périodiquement si le fichier est mis à jour.

`modified`()

Indique que le fichier `robots.txt` a été téléchargé pour la dernière fois au temps courant.

`crawl_delay`(*useragent*)

Renvoie la valeur du paramètre `Crawl-delay` du `robots.txt` pour le *useragent* en question. S'il n'y a pas de tel paramètre ou qu'il ne s'applique pas au *useragent* spécifié ou si l'entrée du `robots.txt` pour ce paramètre a une syntaxe invalide, renvoie `None`.

Ajouté dans la version 3.6.

`request_rate`(*useragent*)

Renvoie le contenu du paramètre `Request-rate` du `robots.txt` sous la forme d'un *named tuple* `RequestRate(requests, seconds)`. S'il n'y a pas de tel paramètre ou qu'il ne s'applique pas au *useragent* spécifié ou si l'entrée du `robots.txt` pour ce paramètre a une syntaxe invalide, `None` est renvoyé.

Ajouté dans la version 3.6.

`site_maps`()

Renvoie le contenu du paramètre de `Sitemap` depuis `robots.txt` dans la forme d'une `list()`. S'il n'y a pas de tel paramètre ou qu'il ne s'applique pas au *useragent* spécifié ou si l'entrée du `robots.txt` pour ce paramètre a une syntaxe invalide, renvoie `None`.

Ajouté dans la version 3.8.

L'exemple suivant présente une utilisation basique de la classe `RobotFileParser` :

```
>>> import urllib.robotparser
>>> rp = urllib.robotparser.RobotFileParser()
>>> rp.set_url("http://www.musi-cal.com/robots.txt")
>>> rp.read()
>>> rrate = rp.request_rate("*")
>>> rrate.requests
3
>>> rrate.seconds
20
>>> rp.crawl_delay("*")
6
>>> rp.can_fetch("*", "http://www.musi-cal.com/cgi-bin/search?city=San+Francisco")
False
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> rp.can_fetch("*", "http://www.musi-cal.com/")
True
```

22.9 http --- HTTP modules

Code source : Lib/http/__init__.py

---

http est un paquet qui rassemble plusieurs modules servant à travailler avec le protocole HTTP (*HyperText Transfer Procotol*) :

- Le module *http.client* est un client HTTP bas niveau. Pour accéder à des ressources web, utiliser le module haut niveau *urllib.request*
- Le module *http.server* contient des classes serveur HTTP basiques basées sur *socketserver*
- Le module *http.cookies* contient des utilitaires liés à la gestion d'état HTTP via les cookies
- Le module *http.cookiejar* fournit un mécanisme de persistance des cookies

The *http* module also defines the following enums that help you work with http related code :

class http.**HTTPStatus**

Ajouté dans la version 3.5.

Sous-classe de *enum.IntEnum* qui définit un ensemble de codes d'état HTTP, messages explicatifs et descriptions complètes écrites en anglais.

Utilisation :

```
>>> from http import HTTPStatus
>>> HTTPStatus.OK
HTTPStatus.OK
>>> HTTPStatus.OK == 200
True
>>> HTTPStatus.OK.value
200
>>> HTTPStatus.OK.phrase
'OK'
>>> HTTPStatus.OK.description
'Request fulfilled, document follows'
>>> list(HTTPStatus)
[HTTPStatus.CONTINUE, HTTPStatus.SWITCHING_PROTOCOLS, ...]
```

22.9.1 Codes d'état HTTP

Supported, IANA-registered status codes available in *http.HTTPStatus* are :

Code	Message	Détails
100	CONTINUE	HTTP Semantics **RFC 9110**, Section 15.2.1
101	SWITCHING_PROTOCOLS	HTTP Semantics **RFC 9110**, Section 15.2.2
102	PROCESSING	*WebDAV* **RFC 2518**, Section 10.1
103	EARLY_HINTS	An HTTP Status Code for Indicating Hints **RFC 8297**
200	OK	HTTP Semantics **RFC 9110**, Section 15.3.1
201	CREATED	HTTP Semantics **RFC 9110**, Section 15.3.2
202	ACCEPTED	HTTP Semantics **RFC 9110**, Section 15.3.3
203	NON_AUTHORITATIVE_INFORMATION	HTTP Semantics **RFC 9110**, Section 15.3.4
204	NO_CONTENT	HTTP Semantics **RFC 9110**, Section 15.3.5
205	RESET_CONTENT	HTTP Semantics **RFC 9110**, Section 15.3.6

suite sur la page suiv

Tableau 1 – suite de la page précédente

Code	Message	Détails
206	PARTIAL_CONTENT	HTTP Semantics **RFC 9110**, Section 15.3.7
207	MULTI_STATUS	*WebDAV* **RFC 4918**, Section 11.1
208	ALREADY_REPORTED	*WebDAV* Binding Extensions **RFC 5842**, Section 7.1 (Expérimental)
226	IM_USED	Delta Encoding in HTTP **RFC 3229**, Section 10.4.1
300	MULTIPLE_CHOICES	HTTP Semantics **RFC 9110**, Section 15.4.1
301	MOVED_PERMANENTLY	HTTP Semantics **RFC 9110**, Section 15.4.2
302	FOUND	HTTP Semantics **RFC 9110**, Section 15.4.3
303	SEE_OTHER	HTTP Semantics **RFC 9110**, Section 15.4.4
304	NOT_MODIFIED	HTTP Semantics **RFC 9110**, Section 15.4.5
305	USE_PROXY	HTTP Semantics **RFC 9110**, Section 15.4.6
307	TEMPORARY_REDIRECT	HTTP Semantics **RFC 9110**, Section 15.4.8
308	PERMANENT_REDIRECT	HTTP Semantics **RFC 9110**, Section 15.4.9
400	BAD_REQUEST	HTTP Semantics **RFC 9110**, Section 15.5.1
401	UNAUTHORIZED	HTTP Semantics **RFC 9110**, Section 15.5.2
402	PAYMENT_REQUIRED	HTTP Semantics **RFC 9110**, Section 15.5.3
403	FORBIDDEN	HTTP Semantics **RFC 9110**, Section 15.5.4
404	NOT_FOUND	HTTP Semantics **RFC 9110**, Section 15.5.5
405	METHOD_NOT_ALLOWED	HTTP Semantics **RFC 9110**, Section 15.5.6
406	NOT_ACCEPTABLE	HTTP Semantics **RFC 9110**, Section 15.5.7
407	PROXY_AUTHENTICATION_REQUIRED	HTTP Semantics **RFC 9110**, Section 15.5.8
408	REQUEST_TIMEOUT	HTTP Semantics **RFC 9110**, Section 15.5.9
409	CONFLICT	HTTP Semantics **RFC 9110**, Section 15.5.10
410	GONE	HTTP Semantics **RFC 9110**, Section 15.5.11
411	LENGTH_REQUIRED	HTTP Semantics **RFC 9110**, Section 15.5.12
412	PRECONDITION_FAILED	HTTP Semantics **RFC 9110**, Section 15.5.13
413	CONTENT_TOO_LARGE	HTTP Semantics **RFC 9110**, Section 15.5.14
414	URI_TOO_LONG	HTTP Semantics **RFC 9110**, Section 15.5.15
415	UNSUPPORTED_MEDIA_TYPE	HTTP Semantics **RFC 9110**, Section 15.5.16
416	RANGE_NOT_SATISFIABLE	HTTP Semantics **RFC 9110**, Section 15.5.17
417	EXPECTATION_FAILED	HTTP Semantics **RFC 9110**, Section 15.5.18
418	IM_A_TEAPOT	HTCPCP/1.0 **RFC 2324**, Section 2.3.2
421	MISDIRECTED_REQUEST	HTTP Semantics **RFC 9110**, Section 15.5.20
422	UNPROCESSABLE_CONTENT	HTTP Semantics **RFC 9110**, Section 15.5.21
423	LOCKED	*WebDAV* **RFC 4918**, Section 11.3
424	FAILED_DEPENDENCY	*WebDAV* **RFC 4918**, Section 11.4
425	TOO_EARLY	Using Early Data in HTTP **RFC 8470**
426	UPGRADE_REQUIRED	HTTP Semantics **RFC 9110**, Section 15.5.22
428	PRECONDITION_REQUIRED	Additional HTTP Status Codes **RFC 6585**
429	TOO_MANY_REQUESTS	Additional HTTP Status Codes **RFC 6585**
431	REQUEST_HEADER_FIELDS_TOO_LARGE	Additional HTTP Status Codes **RFC 6585**
451	UNAVAILABLE_FOR_LEGAL_REASONS	An HTTP Status Code to Report Legal Obstacles **RFC 7725**
500	INTERNAL_SERVER_ERROR	HTTP Semantics **RFC 9110**, Section 15.6.1
501	NOT_IMPLEMENTED	HTTP Semantics **RFC 9110**, Section 15.6.2
502	BAD_GATEWAY	HTTP Semantics **RFC 9110**, Section 15.6.3
503	SERVICE_UNAVAILABLE	HTTP Semantics **RFC 9110**, Section 15.6.4
504	GATEWAY_TIMEOUT	HTTP Semantics **RFC 9110**, Section 15.6.5
505	HTTP_VERSION_NOT_SUPPORTED	HTTP Semantics **RFC 9110**, Section 15.6.6
506	VARIANT_ALSO_NEGOTIATES	Transparent Content Negotiation in HTTP **RFC 2295**, Section 8.1 (Expérime
507	INSUFFICIENT_STORAGE	*WebDAV* **RFC 4918**, Section 11.5
508	LOOP_DETECTED	*WebDAV Binding Extensions* **RFC 5842**, Section 7.2 (Expérimental)
510	NOT_EXTENDED	An HTTP Extension Framework **RFC 2774**, Section 7 (Expérimental)
511	NETWORK_AUTHENTICATION_REQUIRED	Codes d'état HTTP supplémentaires **RFC 6585**, Section 6

Dans le but de préserver la compatibilité descendante, les valeurs d'énumération sont aussi présentes dans le module `http.client` sous forme de constantes. Les noms de valeurs de l'énumération sont accessibles de deux manières :

par exemple, le code HTTP 200 est accessible sous les noms `http.HTTPStatus.OK` et `http.client.OK`.

Modifié dans la version 3.7 : Ajouté le code d'état `421 MISDIRECTED_REQUEST`.

Ajouté dans la version 3.8 : Added `451 UNAVAILABLE_FOR_LEGAL_REASONS` status code.

Ajouté dans la version 3.9 : Added `103 EARLY_HINTS`, `418 IM_A_TEAPOT` and `425 TOO_EARLY` status codes.

Modifié dans la version 3.13 : Implemented RFC9110 naming for status constants. Old constant names are preserved for backwards compatibility.

22.9.2 HTTP status category

Ajouté dans la version 3.12.

The enum values have several properties to indicate the HTTP status category :

Property	Indicates that	Détails
`is_informational`	`100 <= status <= 199`	HTTP Semantics **RFC 9110**, Section 15
`is_success`	`200 <= status <= 299`	HTTP Semantics **RFC 9110**, Section 15
`is_redirection`	`300 <= status <= 399`	HTTP Semantics **RFC 9110**, Section 15
`is_client_error`	`400 <= status <= 499`	HTTP Semantics **RFC 9110**, Section 15
`is_server_error`	`500 <= status <= 599`	HTTP Semantics **RFC 9110**, Section 15

Utilisation :

```
>>> from http import HTTPStatus
>>> HTTPStatus.OK.is_success
True
>>> HTTPStatus.OK.is_client_error
False
```

class `http.`**`HTTPMethod`**

Ajouté dans la version 3.11.

A subclass of *`enum.StrEnum`* that defines a set of HTTP methods and descriptions written in English.

Utilisation :

```
>>> from http import HTTPMethod
>>>
>>> HTTPMethod.GET
<HTTPMethod.GET>
>>> HTTPMethod.GET == 'GET'
True
>>> HTTPMethod.GET.value
'GET'
>>> HTTPMethod.GET.description
'Retrieve the target.'
>>> list(HTTPMethod)
[<HTTPMethod.CONNECT>,
 <HTTPMethod.DELETE>,
 <HTTPMethod.GET>,
 <HTTPMethod.HEAD>,
 <HTTPMethod.OPTIONS>,
 <HTTPMethod.PATCH>,
 <HTTPMethod.POST>,
 <HTTPMethod.PUT>,
 <HTTPMethod.TRACE>]
```

22.9.3 HTTP methods

Supported, IANA-registered methods available in `http.HTTPMethod` are :

Méthode	Message	Détails
GET	GET	HTTP Semantics **RFC 9110**, Section 9.3.1
HEAD	HEAD	HTTP Semantics **RFC 9110**, Section 9.3.2
POST	POST	HTTP Semantics **RFC 9110**, Section 9.3.3
PUT	PUT	HTTP Semantics **RFC 9110**, Section 9.3.4
DELETE	DELETE	HTTP Semantics **RFC 9110**, Section 9.3.5
CONNECT	CONNECT	HTTP Semantics **RFC 9110**, Section 9.3.6
OPTIONS	OPTIONS	HTTP Semantics **RFC 9110**, Section 9.3.7
TRACE	TRACE	HTTP Semantics **RFC 9110**, Section 9.3.8
PATCH	PATCH	HTTP/1.1 **RFC 5789**

22.10 `http.client` --- HTTP protocol client

Code source : Lib/http/client.py

This module defines classes that implement the client side of the HTTP and HTTPS protocols. It is normally not used directly --- the module `urllib.request` uses it to handle URLs that use HTTP and HTTPS.

> **Voir aussi**
>
> The Requests package is recommended for a higher-level HTTP client interface.

> **Note**
>
> L'implémentation d'HTTPS n'est disponible que si Python a été compilé avec la prise en charge de SSL (au moyen du module `ssl`).

Availability : not WASI.

This module does not work or is not available on WebAssembly. See *Plateformes WebAssembly* for more information.

Le module fournit les classes suivantes :

class `http.client.`**`HTTPConnection`**(*host*, *port=None*, [*timeout*,]*source_address=None*, *blocksize=8192*)

An `HTTPConnection` instance represents one transaction with an HTTP server. It should be instantiated by passing it a host and optional port number. If no port number is passed, the port is extracted from the host string if it has the form `host:port`, else the default HTTP port (80) is used. If the optional *timeout* parameter is given, blocking operations (like connection attempts) will timeout after that many seconds (if it is not given, the global default timeout setting is used). The optional *source_address* parameter may be a tuple of a (host, port) to use as the source address the HTTP connection is made from. The optional *blocksize* parameter sets the buffer size in bytes for sending a file-like message body.

Par exemple, tous les appels suivants créent des instances qui se connectent à un serveur sur le même hôte et le même port :

```
>>> h1 = http.client.HTTPConnection('www.python.org')
>>> h2 = http.client.HTTPConnection('www.python.org:80')
>>> h3 = http.client.HTTPConnection('www.python.org', 80)
>>> h4 = http.client.HTTPConnection('www.python.org', 80, timeout=10)
```

Modifié dans la version 3.2 : Le paramètre *source_address* a été ajouté.

Modifié dans la version 3.4 : The *strict* parameter was removed. HTTP 0.9-style "Simple Responses" are no longer supported.

Modifié dans la version 3.7 : Le paramètre *blocksize* a été ajouté.

class `http.client.`**`HTTPSConnection`**(*host*, *port=None*, *, [*timeout*,]*source_address=None*, *context=None*, *blocksize=8192*)

Cette sous-classe de `HTTPConnection` utilise SSL pour communiquer avec des serveurs sécurisés. Le port par défaut est `443`. Si le paramètre *context* est fourni, ce doit être une instance de `ssl.SSLContext` décrivant les diverses options SSL.

Il est conseillé de lire *Security considerations* pour plus d'informations sur les bonnes pratiques.

Modifié dans la version 3.2 : Les paramètres *source_address*, *context* et *check_hostname* ont été ajoutés.

Modifié dans la version 3.2 : This class now supports HTTPS virtual hosts if possible (that is, if `ssl.HAS_SNI` is true).

Modifié dans la version 3.4 : Le paramètre *strict* a été supprimé. Les « réponses simples » de HTTP 0.9 ne sont plus prises en charge.

Modifié dans la version 3.4.3 : This class now performs all the necessary certificate and hostname checks by default. To revert to the previous, unverified, behavior `ssl._create_unverified_context()` can be passed to the *context* parameter.

Modifié dans la version 3.8 : Cette classe sélectionne désormais TLS 1.3 pour `ssl.SSLContext.post_handshake_auth` dans le *context* par défaut ou quand *cert_file* est fourni avec une valeur de *context* personnalisée.

Modifié dans la version 3.10 : This class now sends an ALPN extension with protocol indicator `http/1.1` when no *context* is given. Custom *context* should set ALPN protocols with `set_alpn_protocols()`.

Modifié dans la version 3.12 : The deprecated *key_file*, *cert_file* and *check_hostname* parameters have been removed.

class `http.client.`**`HTTPResponse`**(*sock*, *debuglevel=0*, *method=None*, *url=None*)

Classe dont les instances sont renvoyées dès qu'une connexion est établie. Cette classe n'est jamais instanciée directement par l'utilisateur.

Modifié dans la version 3.4 : Le paramètre *strict* a été supprimé. Les « réponses simples » de HTTP 0.9 ne sont plus prises en charge.

Ce module fournit les fonctions suivantes :

`http.client.`**`parse_headers`**(*fp*)

Parse the headers from a file pointer *fp* representing a HTTP request/response. The file has to be a `BufferedIOBase` reader (i.e. not text) and must provide a valid **RFC 2822** style header.

Cette fonction renvoie une instance de `http.client.HTTPMessage` qui contient les champs d'en-tête, mais pas la charge utile (de même que `HTTPResponse.msg` et `http.server.BaseHTTPRequestHandler.headers`). Après le retour, le pointeur de fichier *fp* est prêt à lire le corps HTTP.

> **Note**
>
> La méthode `parse_headers()` n'analyse pas la ligne initiale d'un message HTTP ; elle n'analyse que les lignes `Name: value`. Le fichier doit être prêt à lire ces lignes de champs, aussi la première ligne doit déjà avoir été consommée avant l'appel de la fonction.

Les exceptions suivantes sont levées selon les cas :

exception `http.client.`**`HTTPException`**

La classe de base des autres exceptions de ce module. C'est une sous-classe de `Exception`.

exception http.client.**NotConnected**

Sous-classe de *HTTPException*.

exception http.client.**InvalidURL**

Sous-classe de *HTTPException*, levée si le port donné n'est pas numérique ou est vide.

exception http.client.**UnknownProtocol**

Sous-classe de *HTTPException*.

exception http.client.**UnknownTransferEncoding**

Sous-classe de *HTTPException*.

exception http.client.**UnimplementedFileMode**

Sous-classe de *HTTPException*.

exception http.client.**IncompleteRead**

Sous-classe de *HTTPException*.

exception http.client.**ImproperConnectionState**

Sous-classe de *HTTPException*.

exception http.client.**CannotSendRequest**

Sous-classe de *ImproperConnectionState*.

exception http.client.**CannotSendHeader**

Sous-classe de *ImproperConnectionState*.

exception http.client.**ResponseNotReady**

Sous-classe de *ImproperConnectionState*.

exception http.client.**BadStatusLine**

Sous-classe de *HTTPException*. Levée si un serveur répond avec un code d'état HTTP qui n'est pas compris.

exception http.client.**LineTooLong**

Sous-classe de *HTTPException*. Levée si une ligne du protocole HTTP est excessivement longue dans ce qui provient du serveur.

exception http.client.**RemoteDisconnected**

Sous-classe de *ConnectionResetError* et *BadStatusLine*. Levée par la méthode *HTTPConnection.getresponse()* quand la tentative de lire la réponse n'aboutit à aucune donnée provenant de la connexion, indiquant ainsi que la partie distante a fermé celle-ci.

Ajouté dans la version 3.5 : Antérieurement, *BadStatusLine*('') était levée.

Les constantes définies dans ce module sont :

http.client.**HTTP_PORT**

Le port par défaut du protocole HTTP (toujours 80).

http.client.**HTTPS_PORT**

Le port par défaut du protocole HTTPS (toujours 443).

http.client.**responses**

Ce dictionnaire associe les codes d'états HTTP 1.1 à leurs noms tels que définis par le W3C.

Par exemple : http.client.responses[http.client.NOT_FOUND] est 'Not Found'.

Voir *Codes d'état HTTP* pour une liste des codes d'état HTTP qui sont disponibles comme constantes dans ce module.

22.10.1 Les objets HTTPConnection

Les instances de la classe *HTTPConnection* possèdent les méthodes suivantes :

`HTTPConnection.`**`request`**(*method*, *url*, *body=None*, *headers={}*, ***, *encode_chunked=False*)

This will send a request to the server using the HTTP request method *method* and the request URI *url*. The provided *url* must be an absolute path to conform with **RFC 2616 §5.1.2** (unless connecting to an HTTP proxy server or using the `OPTIONS` or `CONNECT` methods).

Si *body* est passé en paramètre, les données transmises sont envoyées à la suite des en-têtes. Ce paramètre peut-être une `str`, un *objet octet-comptatible*, un *objet fichier* préalablement ouvert, ou un itérable de classe `bytes`. Si *body* est une chaîne, elle est encodée en ISO-8859-1, valeur par défaut pour HTTP. Si c'est un objet octet-compatible, les octets sont envoyés tels quels. Si c'est un *objet fichier*, le contenu du fichier est envoyé ; cet objet fichier doit implémenter au moins la méthode `read()`. Si l'objet fichier est une instance de `io.TextIOBase`, les données renvoyées par la méthode `read()` sont encodées en ISO-8859-1, sinon les données renvoyées par `read()` sont envoyées telles quelles. Si *body* est un itérable, les éléments de cet itérable sont envoyés jusqu'à ce que l'itérable soit vide.

The *headers* argument should be a mapping of extra HTTP headers to send with the request. A **Host header** must be provided to conform with **RFC 2616 §5.1.2** (unless connecting to an HTTP proxy server or using the `OPTIONS` or `CONNECT` methods).

Si *headers* ne contient ni `Content-Length:` ni `Transfer-Encoding:`, mais qu'un corps de requête est fourni, un de ces en-têtes est ajouté automatiquement. Si *body* est `None`, l'en-tête `Content-Length:` est paramétré à `0` pour les méthodes qui attendent un corps (`PUT`, `POST`, et `PATCH`). Si *body* est une chaîne ou un objet de type octets qui n'est pas un (*objet fichier*), l'en-tête `Content-Length` est paramétré à sa longueur. Un *body* de tout autre type (fichiers ou itérables en général) est encodé par morceaux et l'en-tête `Transfer-Encoding` est automatiquement paramétré à la place de `Content-Length`.

L'argument *encode_chunked* n'est pertinent que si l'en-tête `Transfer-Encoding:` est présent. Si *encode_chunked* est `False`, l'objet HTTPConnection suppose que l'encodage est géré par le code d'appel. S'il vaut `True`, le corps est encodé par morceaux.

For example, to perform a `GET` request to `https://docs.python.org/3/` :

```
>>> import http.client
>>> host = "docs.python.org"
>>> conn = http.client.HTTPSConnection(host)
>>> conn.request("GET", "/3/", headers={"Host": host})
>>> response = conn.getresponse()
>>> print(response.status, response.reason)
200 OK
```

Note

L'encodage pour les transferts par morceaux a été ajouté à la version 1.1 du protocole HTTP. À moins que le serveur HTTP sache gérer HTTP 1.1, l'appelant doit soit spécifier l'en-tête `Content-Length:`, soit passer la représentation du corps de message dans un objet de classe `str` ou un objet octet-compatible qui ne soit pas un fichier.

Modifié dans la version 3.2 : *body* peut désormais être un itérable.

Modifié dans la version 3.6 : Si parmi les en-têtes ne figure ni `Content-Length:`, ni `Transfer-Encoding:`, les objets fichiers et itérables *body* sont désormais encodés par morceaux. L'argument *encode_chunked* a été ajouté. Aucune tentative n'est faite pour essayer de déterminer la valeur de l'en-tête `Content-Length` pour un objet fichier.

`HTTPConnection.`**`getresponse`**()

Doit être appelé après qu'une requête a été envoyée pour récupérer la réponse du serveur. Renvoie une instance de `HTTPResponse`.

Note

> Notez que la totalité de la réponse doit être lue avant de pouvoir envoyer une nouvelle requête au serveur.

Modifié dans la version 3.5 : Si une exception `ConnectionError` ou une de ses sous-classes est levée, l'objet de classe `HTTPConnection` sera prêt à se reconnecter quand une nouvelle requête sera envoyée.

`HTTPConnection.`**`set_debuglevel`**(*level*)

Règle le niveau de débogage. Le niveau de débogage par défaut est `0` (aucune sortie de débogage n'est affichée). Toute valeur plus grande que `0` provoque l'affichage sur `sys.stdout` de toutes les sorties de débogage actuellement définies. Le paramètre `debuglevel` est passé à tout nouvel objet de classe `HTTPResponse` qui est créé.

Ajouté dans la version 3.1.

`HTTPConnection.`**`set_tunnel`**(*host*, *port=None*, *headers=None*)

Paramètre l'hôte et le port pour le tunnel de connexion HTTP. Il permet de réaliser la connexion au travers d'un serveur mandataire.

The *host* and *port* arguments specify the endpoint of the tunneled connection (i.e. the address included in the CONNECT request, *not* the address of the proxy server).

The *headers* argument should be a mapping of extra HTTP headers to send with the CONNECT request.

As HTTP/1.1 is used for HTTP CONNECT tunnelling request, as per the RFC, a HTTP `Host:` header must be provided, matching the authority-form of the request target provided as the destination for the CONNECT request. If a HTTP `Host:` header is not provided via the headers argument, one is generated and transmitted automatically.

Par exemple, pour un tunnel traversant un serveur mandataire HTTPS accessible localement sur le port 8080, nous devons passer l'adresse du serveur mandataire au constructeur de la classe `HTTPSConnection`, en plus de l'éventuelle adresse de l'hôte que nous voulons atteindre, qui elle doit être passée à la méthode `set_tunnel()` :

```
>>> import http.client
>>> conn = http.client.HTTPSConnection("localhost", 8080)
>>> conn.set_tunnel("www.python.org")
>>> conn.request("HEAD","/index.html")
```

Ajouté dans la version 3.2.

Modifié dans la version 3.12 : HTTP CONNECT tunnelling requests use protocol HTTP/1.1, upgraded from protocol HTTP/1.0. `Host:` HTTP headers are mandatory for HTTP/1.1, so one will be automatically generated and transmitted if not provided in the headers argument.

`HTTPConnection.`**`get_proxy_response_headers`**()

Returns a dictionary with the headers of the response received from the proxy server to the CONNECT request.

If the CONNECT request was not sent, the method returns `None`.

Ajouté dans la version 3.12.

`HTTPConnection.`**`connect`**()

Se connecte au serveur spécifié quand l'objet est créé. Par défaut, est appelée automatiquement lorsqu'une requête est faite alors que le client ne s'est pas connecté au préalable.

Lève un *évènement d'audit* `http.client.connect` contenant les arguments `self`, `host`, `port`.

`HTTPConnection.`**`close`**()

Ferme la connexion au serveur.

`HTTPConnection.`**`blocksize`**

Taille en octets du tampon utilisé pour transmettre un corps de message de type fichier.

Ajouté dans la version 3.7.

As an alternative to using the `request()` method described above, you can also send your request step by step, by using the four functions below.

`HTTPConnection.putrequest`(*method*, *url*, *skip_host=False*, *skip_accept_encoding=False*)

Ce doit être le premier appel une fois que la connexion au serveur a été réalisée. Est envoyée au serveur une ligne consistant en la chaîne *method*, la chaîne d'*url*, et la version du protocole HTTP (`HTTP/1.1`). Pour désactiver l'envoi automatique des en-têtes `Host:` ou `Accept-Encoding:` (par exemple pour accepter des encodages supplémentaires de contenus), il est nécessaire de passer les paramètres *skip_host* ou *skip_accept_encoding* avec des valeurs différentes de `False`.

`HTTPConnection.putheader`(*header*, *argument*[, ...])

Envoie un en-tête de style **RFC 822** au serveur. Elle envoie au serveur une ligne regroupant un en-tête, une espace, un `:`, une espace et le premier argument. Si plusieurs arguments sont donnés, des lignes de continuation sont envoyées, chacune d'elle étant constituée d'une espace suivie d'un argument.

`HTTPConnection.endheaders`(*message_body=None*, *, *encode_chunked=False*)

Envoie une ligne blanche au serveur, signalant la fin des en-têtes. L'argument optionnel *message_body* peut-être utilisé pour passer le corps du message associé à la requête.

Si *encode_chunked* est `True`, le résultat de chaque itération sur *message_body* est un morceau encodé selon la **RFC 7230**, section 3.3.1. La façon dont les données sont encodées dépend du type de *message_body*. Si *message_body* implémente l'interface tampon l'encodage donne un unique morceau. Si *message_body* est un itérable de classe `collections.abc.Iterable`, chaque itération sur *message_body* donne un morceau. Si *message_body* est un *fichier objet*, chaque appel à `.read()` renvoie un morceau. La méthode signale automatiquement la fin des données encodées par morceaux immédiatement après *message_body*.

> **Note**
>
> Selon la spécification de l'encodage des morceaux, les morceaux vides renvoyés par un itérateur associé au corps du message sont ignorés par l'encodeur de morceaux. Ce comportement est choisi pour éviter que la lecture de la requête par le serveur cible ne se termine prématurément pour cause d'encodage mal formé.

Modifié dans la version 3.6 : Added chunked encoding support and the *encode_chunked* parameter.

`HTTPConnection.send`(*data*)

Envoie les données au serveur. Elle ne peut être utilisée directement qu'une fois la méthode `endheaders()` a été appelée et avant que la méthode `getresponse()` ait été appelée.

Lève un *évènement d'audit* `http.client.send` avec comme arguments `self`, `data`.

22.10.2 Les objets HTTPResponse

Une instance de `HTTPResponse` encapsule la réponse HTTP du serveur. Elle fournit un accès aux en-têtes et au corps de la réponse. La réponse est un objet itérable pouvant être utilisé avec l'instruction `with`.

Modifié dans la version 3.5 : L'interface de la classe `io.BufferedIOBase` est désormais implémentée et toutes ses opérations de lecture sont gérées.

`HTTPResponse.read`([*amt*])

Lit et renvoie soit tout le corps de la réponse soit une partie de celui-ci se limitant aux *amt* octets suivants.

`HTTPResponse.readinto`(*b*)

Lit les prochains `len(b)` octets du corps de la réponse et les place dans le tampon *b*. Renvoie le nombre d'octets lus.

Ajouté dans la version 3.3.

`HTTPResponse.getheader`(*name*, *default=None*)

Return the value of the header *name*, or *default* if there is no header matching *name*. If there is more than one header with the name *name*, return all of the values joined by ', '. If *default* is any iterable other than a single string, its elements are similarly returned joined by commas.

`HTTPResponse.getheaders`()

Renvoie une liste d'*n*-uplets `(en-tête, valeur)`.

`HTTPResponse.fileno()`

Renvoie le `fileno` du connecteur réseau sous-jacent.

`HTTPResponse.msg`

Une instance de *http.client.HTTPMessage* contenant les en-têtes de la réponse. La classe *http.client.HTTPMessage* est une sous-classe de la classe *email.message.Message*.

`HTTPResponse.version`

La version du protocole HTTP utilisée par le serveur : 10 pour HTTP/1.0, 11 pour HTTP/1.1.

`HTTPResponse.url`

L'URL de la ressource récupérée, utilisée habituellement pour déterminer si une redirection a été suivie.

`HTTPResponse.headers`

Les en-têtes de la réponse sous la forme d'une instance de *email.message.EmailMessage*.

`HTTPResponse.status`

Code d'état renvoyé par le serveur.

`HTTPResponse.reason`

Phrase renvoyée par le serveur et indiquant la cause.

`HTTPResponse.debuglevel`

Un point d'entrée pour débogage. Si *debuglevel* est plus grand que zéro, les messages sont envoyés à `sys.stdout` pendant la lecture et l'analyse de la réponse.

`HTTPResponse.closed`

Vaut `True` si le flux est terminé.

`HTTPResponse.geturl()`

Obsolète depuis la version 3.9 : Rendu obsolète par *url*.

`HTTPResponse.info()`

Obsolète depuis la version 3.9 : Rendu obsolète par *headers*.

`HTTPResponse.getcode()`

Obsolète depuis la version 3.9 : Rendu obsolète par *status*.

22.10.3 Exemples

Voici un exemple de session utilisant la méthode `GET` :

```
>>> import http.client
>>> conn = http.client.HTTPSConnection("www.python.org")
>>> conn.request("GET", "/")
>>> r1 = conn.getresponse()
>>> print(r1.status, r1.reason)
200 OK
>>> data1 = r1.read()  # This will return entire content.
>>> # The following example demonstrates reading data in chunks.
>>> conn.request("GET", "/")
>>> r1 = conn.getresponse()
>>> while chunk := r1.read(200):
...     print(repr(chunk))
b'<!doctype html>\n<!--[if"...
...
>>> # Example of an invalid request
>>> conn = http.client.HTTPSConnection("docs.python.org")
>>> conn.request("GET", "/parrot.spam")
>>> r2 = conn.getresponse()
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> print(r2.status, r2.reason)
404 Not Found
>>> data2 = r2.read()
>>> conn.close()
```

Voici un exemple de session utilisant la méthode `HEAD`. Notez que la méthode `HEAD` ne renvoie jamais de données.

```
>>> import http.client
>>> conn = http.client.HTTPSConnection("www.python.org")
>>> conn.request("HEAD", "/")
>>> res = conn.getresponse()
>>> print(res.status, res.reason)
200 OK
>>> data = res.read()
>>> print(len(data))
0
>>> data == b''
True
```

Here is an example session that uses the `POST` method :

```
>>> import http.client, urllib.parse
>>> params = urllib.parse.urlencode({'@number': 12524, '@type': 'issue', '@action
↪': 'show'})
>>> headers = {"Content-type": "application/x-www-form-urlencoded",
...            "Accept": "text/plain"}
>>> conn = http.client.HTTPConnection("bugs.python.org")
>>> conn.request("POST", "", params, headers)
>>> response = conn.getresponse()
>>> print(response.status, response.reason)
302 Found
>>> data = response.read()
>>> data
b'Redirecting to <a href="https://bugs.python.org/issue12524">https://bugs.python.
↪org/issue12524</a>'
>>> conn.close()
```

Client side HTTP `PUT` requests are very similar to `POST` requests. The difference lies only on the server side where HTTP servers will allow resources to be created via `PUT` requests. It should be noted that custom HTTP methods are also handled in *`urllib.request.Request`* by setting the appropriate method attribute. Here is an example session that uses the `PUT` method :

```
>>> # This creates an HTTP request
>>> # with the content of BODY as the enclosed representation
>>> # for the resource http://localhost:8080/file
...
>>> import http.client
>>> BODY = "***filecontents***"
>>> conn = http.client.HTTPConnection("localhost", 8080)
>>> conn.request("PUT", "/file", BODY)
>>> response = conn.getresponse()
>>> print(response.status, response.reason)
200, OK
```

22.10.4 Les objets HTTPMessage

class `http.client.`**`HTTPMessage`**(*email.message.Message*)

Une instance de classe `http.client.HTTPMessage` contient les en-têtes d'une réponse HTTP. Elle est implémentée en utilisant la classe `email.message.Message`.

22.11 `ftplib` --- FTP protocol client

Code source : Lib/ftplib.py

Ce module définit la classe `FTP` et quelques éléments associés. La classe `FTP` implémente le côté client du protocole FTP. Vous pouvez l'utiliser pour écrire des programmes Python qui effectuent diverses tâches automatisées telles que la synchronisation d'autres serveurs FTP. Il est aussi utilisé par le module `urllib.request` pour gérer les URL qui utilisent FTP. Pour plus d'informations sur FTP (File Transfer Protocol), voir la **RFC 959**.

L'encodage par défaut est UTF-8, voir la **RFC 2640**.

Availability : not WASI.

This module does not work or is not available on WebAssembly. See *Plateformes WebAssembly* for more information.

Voici un exemple de session utilisant le module `ftplib` :

```
>>> from ftplib import FTP
>>> ftp = FTP('ftp.us.debian.org')   # connect to host, default port
>>> ftp.login()                      # user anonymous, passwd anonymous@
'230 Login successful.'
>>> ftp.cwd('debian')                # change into "debian" directory
'250 Directory successfully changed.'
>>> ftp.retrlines('LIST')            # list directory contents
-rw-rw-r--    1 1176     1176         1063 Jun 15 10:18 README
...
drwxr-sr-x    5 1176     1176         4096 Dec 19  2000 pool
drwxr-sr-x    4 1176     1176         4096 Nov 17  2008 project
drwxr-xr-x    3 1176     1176         4096 Oct 10  2012 tools
'226 Directory send OK.'
>>> with open('README', 'wb') as fp:
>>>     ftp.retrbinary('RETR README', fp.write)
'226 Transfer complete.'
>>> ftp.quit()
'221 Goodbye.'
```

22.11.1 Référence

FTP objects

class `ftplib.`**`FTP`**(*host=''*, *user=''*, *passwd=''*, *acct=''*, *timeout=None*, *source_address=None*, ***, *encoding='utf-8'*)

Return a new instance of the `FTP` class.

Paramètres

- **host** (`str`) -- The hostname to connect to. If given, `connect(host)` is implicitly called by the constructor.
- **user** (`str`) -- The username to log in with (default : `'anonymous'`). If given, `login(host, passwd, acct)` is implicitly called by the constructor.
- **passwd** (`str`) -- The password to use when logging in. If not given, and if *passwd* is the empty string or `"-"`, a password will be automatically generated.
- **acct** (`str`) -- Account information to be used for the `ACCT` FTP command. Few systems implement this. See RFC-959 for more details.

— **timeout** (*float | None*) -- A timeout in seconds for blocking operations like *connect()* (default : the global default timeout setting).
— **source_address** (*tuple | None*) -- A 2-tuple `(host, port)` for the socket to bind to as its source address before connecting.
— **encoding** (*str*) -- The encoding for directories and filenames (default : `'utf-8'`).

La classe *FTP* peut s'utiliser avec l'instruction `with`, p. ex. :

```
>>> from ftplib import FTP
>>> with FTP("ftp1.at.proftpd.org") as ftp:
...     ftp.login()
...     ftp.dir()
...
'230 Anonymous login ok, restrictions apply.'
dr-xr-xr-x   9 ftp       ftp            154 May  6 10:43 .
dr-xr-xr-x   9 ftp       ftp            154 May  6 10:43 ..
dr-xr-xr-x   5 ftp       ftp           4096 May  6 10:43 CentOS
dr-xr-xr-x   3 ftp       ftp             18 Jul 10  2008 Fedora
>>>
```

Modifié dans la version 3.2 : La prise en charge de l'instruction `with` a été ajoutée.

Modifié dans la version 3.3 : *source_address* parameter was added.

Modifié dans la version 3.9 : If the *timeout* parameter is set to be zero, it will raise a *ValueError* to prevent the creation of a non-blocking socket. The *encoding* parameter was added, and the default was changed from Latin-1 to UTF-8 to follow **RFC 2640**.

Several `FTP` methods are available in two flavors : one for handling text files and another for binary files. The methods are named for the command which is used followed by `lines` for the text version or `binary` for the binary version.

Les instances de la classe *FTP* possèdent les méthodes suivantes :

set_debuglevel(*level*)

Set the instance's debugging level as an *int*. This controls the amount of debugging output printed. The debug levels are :
— `0` (default) : No debug output.
— `1` : Produce a moderate amount of debug output, generally a single line per request.
— `2` or higher : Produce the maximum amount of debugging output, logging each line sent and received on the control connection.

connect(*host=''*, *port=0*, *timeout=None*, *source_address=None*)

Connect to the given host and port. This function should be called only once for each instance ; it should not be called if a *host* argument was given when the *FTP* instance was created. All other `FTP` methods can only be called after a connection has successfully been made.

Paramètres
— **host** (*str*) -- The host to connect to.
— **port** (*int*) -- The TCP port to connect to (default : `21`, as specified by the FTP protocol specification). It is rarely needed to specify a different port number.
— **timeout** (*float | None*) -- A timeout in seconds for the connection attempt (default : the global default timeout setting).
— **source_address** (*tuple | None*) -- A 2-tuple `(host, port)` for the socket to bind to as its source address before connecting.

Raises an *auditing event* `ftplib.connect` with arguments `self`, `host`, `port`.

Modifié dans la version 3.3 : *source_address* parameter was added.

getwelcome()

Return the welcome message sent by the server in reply to the initial connection. (This message sometimes contains disclaimers or help information that may be relevant to the user.)

login(*user='anonymous'*, *passwd=''*, *acct=''*)

Log on to the connected FTP server. This function should be called only once for each instance, after a connection has been established ; it should not be called if the *host* and *user* arguments were given when the `FTP` instance was created. Most FTP commands are only allowed after the client has logged in.

Paramètres

- **user** (`str`) -- The username to log in with (default : `'anonymous'`).
- **passwd** (`str`) -- The password to use when logging in. If not given, and if *passwd* is the empty string or `"-"`, a password will be automatically generated.
- **acct** (`str`) -- Account information to be used for the `ACCT` FTP command. Few systems implement this. See RFC-959 for more details.

abort()

Abort a file transfer that is in progress. Using this does not always work, but it's worth a try.

sendcmd(*cmd*)

Send a simple command string to the server and return the response string.

Raises an *auditing event* `ftplib.sendcmd` with arguments `self`, `cmd`.

voidcmd(*cmd*)

Send a simple command string to the server and handle the response. Return the response string if the response code corresponds to success (codes in the range 200--299). Raise `error_reply` otherwise.

Raises an *auditing event* `ftplib.sendcmd` with arguments `self`, `cmd`.

retrbinary(*cmd*, *callback*, *blocksize=8192*, *rest=None*)

Retrieve a file in binary transfer mode.

Paramètres

- **cmd** (`str`) -- An appropriate `RETR` command : `"RETR filename"`.
- **callback** (*callable*) -- A single parameter callable that is called for each block of data received, with its single argument being the data as `bytes`.
- **blocksize** (`int`) -- The maximum chunk size to read on the low-level `socket` object created to do the actual transfer. This also corresponds to the largest size of data that will be passed to *callback*. Defaults to `8192`.
- **rest** (`int`) -- A `REST` command to be sent to the server. See the documentation for the *rest* parameter of the `transfercmd()` method.

retrlines(*cmd*, *callback=None*)

Retrieve a file or directory listing in the encoding specified by the *encoding* parameter at initialization. *cmd* should be an appropriate `RETR` command (see `retrbinary()`) or a command such as `LIST` or `NLST` (usually just the string `'LIST'`). `LIST` retrieves a list of files and information about those files. `NLST` retrieves a list of file names. The *callback* function is called for each line with a string argument containing the line with the trailing CRLF stripped. The default *callback* prints the line to `sys.stdout`.

set_pasv(*val*)

Enable "passive" mode if *val* is true, otherwise disable passive mode. Passive mode is on by default.

storbinary(*cmd*, *fp*, *blocksize=8192*, *callback=None*, *rest=None*)

Store a file in binary transfer mode.

Paramètres

- **cmd** (`str`) -- An appropriate `STOR` command : `"STOR filename"`.
- **fp** (*file object*) -- A file object (opened in binary mode) which is read until EOF, using its `read()` method in blocks of size *blocksize* to provide the data to be stored.
- **blocksize** (`int`) -- The read block size. Defaults to `8192`.
- **callback** (*callable*) -- A single parameter callable that is called for each block of data sent, with its single argument being the data as `bytes`.
- **rest** (`int`) -- A `REST` command to be sent to the server. See the documentation for the *rest* parameter of the `transfercmd()` method.

Modifié dans la version 3.2 : The *rest* parameter was added.

storlines(*cmd*, *fp*, *callback=None*)

Store a file in line mode. *cmd* should be an appropriate `STOR` command (see `storbinary()`). Lines are read until EOF from the *file object fp* (opened in binary mode) using its `readline()` method to provide the data to be stored. *callback* is an optional single parameter callable that is called on each line after it is sent.

transfercmd(*cmd*, *rest=None*)

Initiate a transfer over the data connection. If the transfer is active, send an `EPRT` or `PORT` command and the transfer command specified by *cmd*, and accept the connection. If the server is passive, send an `EPSV` or `PASV` command, connect to it, and start the transfer command. Either way, return the socket for the connection.

If optional *rest* is given, a `REST` command is sent to the server, passing *rest* as an argument. *rest* is usually a byte offset into the requested file, telling the server to restart sending the file's bytes at the requested offset, skipping over the initial bytes. Note however that the `transfercmd()` method converts *rest* to a string with the *encoding* parameter specified at initialization, but no check is performed on the string's contents. If the server does not recognize the `REST` command, an `error_reply` exception will be raised. If this happens, simply call `transfercmd()` without a *rest* argument.

ntransfercmd(*cmd*, *rest=None*)

Like `transfercmd()`, but returns a tuple of the data connection and the expected size of the data. If the expected size could not be computed, `None` will be returned as the expected size. *cmd* and *rest* means the same thing as in `transfercmd()`.

mlsd(*path=''*, *facts=[]*)

List a directory in a standardized format by using `MLSD` command (**RFC 3659**). If *path* is omitted the current directory is assumed. *facts* is a list of strings representing the type of information desired (e.g. `["type", "size", "perm"]`). Return a generator object yielding a tuple of two elements for every file found in path. First element is the file name, the second one is a dictionary containing facts about the file name. Content of this dictionary might be limited by the *facts* argument but server is not guaranteed to return all requested facts.

Ajouté dans la version 3.3.

nlst(*argument*[, ...])

Renvoie une liste de noms de fichiers comme celle que renvoie la commande `NLST`. Le paramètre optionnel *argument* est une liste de dossiers (la valeur par défaut est le répertoire courant du serveur). Plusieurs paramètres peuvent être utilisés pour passer des paramètres non standards à la commande `NLST`.

> **Note**
>
> La commande `mlsd()` offre une meilleure API si votre serveur sait la gérer.

dir(*argument*[, ...])

Produce a directory listing as returned by the `LIST` command, printing it to standard output. The optional *argument* is a directory to list (default is the current server directory). Multiple arguments can be used to pass non-standard options to the `LIST` command. If the last argument is a function, it is used as a *callback* function as for `retrlines()` ; the default prints to `sys.stdout`. This method returns `None`.

> **Note**
>
> La commande `mlsd()` offre une meilleure API si votre serveur sait la gérer.

rename(*fromname*, *toname*)

Renomme le fichier portant le nom *fromname* en *toname* sur le serveur.

delete(*filename*)

Remove the file named *filename* from the server. If successful, returns the text of the response, otherwise

raises `error_perm` on permission errors or `error_reply` on other errors.

cwd(*pathname*)

Set the current directory on the server.

mkd(*pathname*)

Crée un nouveau dossier sur le serveur.

pwd()

Renvoie le chemin d'accès au répertoire courant sur le serveur.

rmd(*dirname*)

Supprime le dossier portant le nom *dirname* sur le serveur.

size(*filename*)

Request the size of the file named *filename* on the server. On success, the size of the file is returned as an integer, otherwise `None` is returned. Note that the `SIZE` command is not standardized, but is supported by many common server implementations.

quit()

Send a `QUIT` command to the server and close the connection. This is the "polite" way to close a connection, but it may raise an exception if the server responds with an error to the `QUIT` command. This implies a call to the `close()` method which renders the `FTP` instance useless for subsequent calls (see below).

close()

Close the connection unilaterally. This should not be applied to an already closed connection such as after a successful call to `quit()`. After this call the `FTP` instance should not be used any more (after a call to `close()` or `quit()` you cannot reopen the connection by issuing another `login()` method).

FTP_TLS objects

class ftplib.**FTP_TLS**(*host=''*, *user=''*, *passwd=''*, *acct=''*, **, context=None*, *timeout=None*, *source_address=None*, *encoding='utf-8'*)

An `FTP` subclass which adds TLS support to FTP as described in **RFC 4217**. Connect to port 21 implicitly securing the FTP control connection before authenticating.

> **Note**
>
> The user must explicitly secure the data connection by calling the `prot_p()` method.

Paramètres

- **host** (`str`) -- The hostname to connect to. If given, `connect(host)` is implicitly called by the constructor.
- **user** (`str`) -- The username to log in with (default : `'anonymous'`). If given, `login(host, passwd, acct)` is implicitly called by the constructor.
- **passwd** (`str`) -- The password to use when logging in. If not given, and if *passwd* is the empty string or `"-"`, a password will be automatically generated.
- **acct** (`str`) -- Account information to be used for the `ACCT` FTP command. Few systems implement this. See RFC-959 for more details.
- **context** (`ssl.SSLContext`) -- An SSL context object which allows bundling SSL configuration options, certificates and private keys into a single, potentially long-lived, structure. Please read *Security considerations* for best practices.
- **timeout** (`float | None`) -- A timeout in seconds for blocking operations like `connect()` (default : the global default timeout setting).
- **source_address** (`tuple | None`) -- A 2-tuple `(host, port)` for the socket to bind to as its source address before connecting.
- **encoding** (`str`) -- The encoding for directories and filenames (default : `'utf-8'`).

Ajouté dans la version 3.2.

Modifié dans la version 3.3 : Added the *source_address* parameter.

Modifié dans la version 3.4 : The class now supports hostname check with `ssl.SSLContext.check_hostname` and *Server Name Indication* (see `ssl.HAS_SNI`).

Modifié dans la version 3.9 : If the *timeout* parameter is set to be zero, it will raise a `ValueError` to prevent the creation of a non-blocking socket. The *encoding* parameter was added, and the default was changed from Latin-1 to UTF-8 to follow **RFC 2640**.

Modifié dans la version 3.12 : The deprecated *keyfile* and *certfile* parameters have been removed.

Voici un exemple de session utilisant la classe `FTP_TLS` :

```
>>> ftps = FTP_TLS('ftp.pureftpd.org')
>>> ftps.login()
'230 Anonymous user logged in'
>>> ftps.prot_p()
'200 Data protection level set to "private"'
>>> ftps.nlst()
['6jack', 'OpenBSD', 'antilink', 'blogbench', 'bsdcam', 'clockspeed', 'djbdns-
↪jedi', 'docs', 'eaccelerator-jedi', 'favicon.ico', 'francotone', 'fugu',
↪'ignore', 'libpuzzle', 'metalog', 'minidentd', 'misc', 'mysql-udf-global-
↪user-variables', 'php-jenkins-hash', 'php-skein-hash', 'php-webdav',
↪'phpaudit', 'phpbench', 'pincaster', 'ping', 'posto', 'pub', 'public',
↪'public_keys', 'pure-ftpd', 'qscan', 'qtc', 'sharedance', 'skycache', 'sound
↪', 'tmp', 'ucarp']
```

`FTP_TLS` class inherits from `FTP`, defining these additional methods and attributes :

ssl_version

The SSL version to use (defaults to `ssl.PROTOCOL_SSLv23`).

auth()

Set up a secure control connection by using TLS or SSL, depending on what is specified in the `ssl_version` attribute.

Modifié dans la version 3.4 : The method now supports hostname check with `ssl.SSLContext.check_hostname` and *Server Name Indication* (see `ssl.HAS_SNI`).

ccc()

Revert control channel back to plaintext. This can be useful to take advantage of firewalls that know how to handle NAT with non-secure FTP without opening fixed ports.

Ajouté dans la version 3.3.

prot_p()

Établit une connexion de données sécurisée.

prot_c()

Établit une connexion de données non sécurisées.

Module variables

exception ftplib.**error_reply**

Exception levée lorsqu'une réponse inattendue est reçue du serveur.

exception ftplib.**error_temp**

Exception levée lorsqu'un code d'erreur signifiant une erreur temporaire (code de réponse dans l'intervalle 400-499) est reçu.

exception ftplib.**error_perm**

Exception levée lorsqu'un code d'erreur signifiant une erreur permanente (code de réponse dans l'intervalle 500-599) est reçu.

exception ftplib.**error_proto**

Exception levée lorsqu'une réponse reçue du serveur ne correspond pas aux à la spécification de File Transfer Protocol, c.-à-d. qu'elle doit commencer par un chiffre dans l'intervalle 1-5.

ftplib.**all_errors**

The set of all exceptions (as a tuple) that methods of `FTP` instances may raise as a result of problems with the FTP connection (as opposed to programming errors made by the caller). This set includes the four exceptions listed above as well as `OSError` and `EOFError`.

Voir aussi

Module ***netrc***

Parser for the `.netrc` file format. The file `.netrc` is typically used by FTP clients to load user authentication information before prompting the user.

22.12 poplib --- POP3 protocol client

Code source : Lib/poplib.py

This module defines a class, `POP3`, which encapsulates a connection to a POP3 server and implements the protocol as defined in **RFC 1939**. The `POP3` class supports both the minimal and optional command sets from **RFC 1939**. The `POP3` class also supports the `STLS` command introduced in **RFC 2595** to enable encrypted communication on an already established connection.

Additionally, this module provides a class `POP3_SSL`, which provides support for connecting to POP3 servers that use SSL as an underlying protocol layer.

Note that POP3, though widely supported, is obsolescent. The implementation quality of POP3 servers varies widely, and too many are quite poor. If your mailserver supports IMAP, you would be better off using the `imaplib.IMAP4` class, as IMAP servers tend to be better implemented.

Availability : not WASI.

This module does not work or is not available on WebAssembly. See *Plateformes WebAssembly* for more information.

The `poplib` module provides two classes :

class poplib.**POP3**(*host*, *port=POP3_PORT*[, *timeout*])

This class implements the actual POP3 protocol. The connection is created when the instance is initialized. If *port* is omitted, the standard POP3 port (110) is used. The optional *timeout* parameter specifies a timeout in seconds for the connection attempt (if not specified, the global default timeout setting will be used).

Raises an *auditing event* `poplib.connect` with arguments `self`, `host`, `port`.

All commands will raise an *auditing event* `poplib.putline` with arguments `self` and `line`, where `line` is the bytes about to be sent to the remote host.

Modifié dans la version 3.9 : If the *timeout* parameter is set to be zero, it will raise a `ValueError` to prevent the creation of a non-blocking socket.

class poplib.**POP3_SSL**(*host*, *port=POP3_SSL_PORT*, **, *timeout=None*, *context=None*)

This is a subclass of `POP3` that connects to the server over an SSL encrypted socket. If *port* is not specified, 995, the standard POP3-over-SSL port is used. *timeout* works as in the `POP3` constructor. *context* is an optional `ssl.SSLContext` object which allows bundling SSL configuration options, certificates and private keys into a single (potentially long-lived) structure. Please read *Security considerations* for best practices.

Raises an *auditing event* `poplib.connect` with arguments `self`, `host`, `port`.

All commands will raise an *auditing event* `poplib.putline` with arguments `self` and `line`, where `line` is the bytes about to be sent to the remote host.

Modifié dans la version 3.2 : *context* parameter added.

Modifié dans la version 3.4 : The class now supports hostname check with `ssl.SSLContext.check_hostname` and *Server Name Indication* (see `ssl.HAS_SNI`).

Modifié dans la version 3.9 : If the *timeout* parameter is set to be zero, it will raise a `ValueError` to prevent the creation of a non-blocking socket.

Modifié dans la version 3.12 : The deprecated *keyfile* and *certfile* parameters have been removed.

One exception is defined as an attribute of the `poplib` module :

exception `poplib.`**`error_proto`**

Exception raised on any errors from this module (errors from `socket` module are not caught). The reason for the exception is passed to the constructor as a string.

Voir aussi

Module `imaplib`
The standard Python IMAP module.

Frequently Asked Questions About Fetchmail
The FAQ for the **`fetchmail`** POP/IMAP client collects information on POP3 server variations and RFC noncompliance that may be useful if you need to write an application based on the POP protocol.

22.12.1 POP3 Objects

All POP3 commands are represented by methods of the same name, in lowercase ; most return the response text sent by the server.

A `POP3` instance has the following methods :

`POP3.`**`set_debuglevel`**(*level*)

Set the instance's debugging level. This controls the amount of debugging output printed. The default, `0`, produces no debugging output. A value of `1` produces a moderate amount of debugging output, generally a single line per request. A value of `2` or higher produces the maximum amount of debugging output, logging each line sent and received on the control connection.

`POP3.`**`getwelcome`**()

Returns the greeting string sent by the POP3 server.

`POP3.`**`capa`**()

Query the server's capabilities as specified in **RFC 2449**. Returns a dictionary in the form `{'name': ['param'...]}`.

Ajouté dans la version 3.4.

`POP3.`**`user`**(*username*)

Send user command, response should indicate that a password is required.

`POP3.`**`pass_`**(*password*)

Send password, response includes message count and mailbox size. Note : the mailbox on the server is locked until `quit()` is called.

`POP3.`**`apop`**(*user*, *secret*)

Use the more secure APOP authentication to log into the POP3 server.

`POP3.`**`rpop`**(*user*)

Use RPOP authentication (similar to UNIX r-commands) to log into POP3 server.

`POP3.stat()`

Get mailbox status. The result is a tuple of 2 integers : `(message count, mailbox size)`.

`POP3.list([which])`

Request message list, result is in the form `(response, ['mesg_num octets', ...], octets)`. If *which* is set, it is the message to list.

`POP3.retr(which)`

Retrieve whole message number *which*, and set its seen flag. Result is in form `(response, ['line', ...], octets)`.

`POP3.dele(which)`

Flag message number *which* for deletion. On most servers deletions are not actually performed until QUIT (the major exception is Eudora QPOP, which deliberately violates the RFCs by doing pending deletes on any disconnect).

`POP3.rset()`

Remove any deletion marks for the mailbox.

`POP3.noop()`

Do nothing. Might be used as a keep-alive.

`POP3.quit()`

Signoff : commit changes, unlock mailbox, drop connection.

`POP3.top(which, howmuch)`

Retrieves the message header plus *howmuch* lines of the message after the header of message number *which*. Result is in form `(response, ['line', ...], octets)`.

The POP3 TOP command this method uses, unlike the RETR command, doesn't set the message's seen flag ; unfortunately, TOP is poorly specified in the RFCs and is frequently broken in off-brand servers. Test this method by hand against the POP3 servers you will use before trusting it.

`POP3.uidl(which=None)`

Return message digest (unique id) list. If *which* is specified, result contains the unique id for that message in the form `'response mesgnum uid`, otherwise result is list `(response, ['mesgnum uid', ...], octets)`.

`POP3.utf8()`

Try to switch to UTF-8 mode. Returns the server response if successful, raises `error_proto` if not. Specified in **RFC 6856**.

Ajouté dans la version 3.5.

`POP3.stls(context=None)`

Start a TLS session on the active connection as specified in **RFC 2595**. This is only allowed before user authentication

context parameter is a `ssl.SSLContext` object which allows bundling SSL configuration options, certificates and private keys into a single (potentially long-lived) structure. Please read *Security considerations* for best practices.

This method supports hostname checking via `ssl.SSLContext.check_hostname` and *Server Name Indication* (see `ssl.HAS_SNI`).

Ajouté dans la version 3.4.

Instances of `POP3_SSL` have no additional methods. The interface of this subclass is identical to its parent.

22.12.2 POP3 Example

Here is a minimal example (without error checking) that opens a mailbox and retrieves and prints all messages :

```
import getpass, poplib

M = poplib.POP3('localhost')
M.user(getpass.getuser())
M.pass_(getpass.getpass())
numMessages = len(M.list()[1])
for i in range(numMessages):
    for j in M.retr(i+1)[1]:
        print(j)
```

At the end of the module, there is a test section that contains a more extensive example of usage.

22.13 `imaplib` --- IMAP4 protocol client

Code source : Lib/imaplib.py

This module defines three classes, *IMAP4*, *IMAP4_SSL* and *IMAP4_stream*, which encapsulate a connection to an IMAP4 server and implement a large subset of the IMAP4rev1 client protocol as defined in **RFC 2060**. It is backward compatible with IMAP4 (**RFC 1730**) servers, but note that the `STATUS` command is not supported in IMAP4.

Availability : not WASI.

This module does not work or is not available on WebAssembly. See *Plateformes WebAssembly* for more information.

Three classes are provided by the *imaplib* module, *IMAP4* is the base class :

class imaplib.**IMAP4**(*host=''*, *port=IMAP4_PORT*, *timeout=None*)

This class implements the actual IMAP4 protocol. The connection is created and protocol version (IMAP4 or IMAP4rev1) is determined when the instance is initialized. If *host* is not specified, `''` (the local host) is used. If *port* is omitted, the standard IMAP4 port (143) is used. The optional *timeout* parameter specifies a timeout in seconds for the connection attempt. If timeout is not given or is `None`, the global default socket timeout is used.

The *IMAP4* class supports the `with` statement. When used like this, the IMAP4 `LOGOUT` command is issued automatically when the `with` statement exits. E.g. :

```
>>> from imaplib import IMAP4
>>> with IMAP4("domain.org") as M:
...     M.noop()
...
('OK', [b'Nothing Accomplished. d25if65hy903weo.87'])
```

Modifié dans la version 3.5 : La prise en charge de l'instruction `with` a été ajoutée.

Modifié dans la version 3.9 : The optional *timeout* parameter was added.

Three exceptions are defined as attributes of the *IMAP4* class :

exception IMAP4.**error**

Exception raised on any errors. The reason for the exception is passed to the constructor as a string.

exception IMAP4.**abort**

IMAP4 server errors cause this exception to be raised. This is a sub-class of *IMAP4.error*. Note that closing the instance and instantiating a new one will usually allow recovery from this exception.

exception IMAP4.**readonly**

This exception is raised when a writable mailbox has its status changed by the server. This is a sub-class of *IMAP4.error*. Some other client now has write permission, and the mailbox will need to be re-opened to re-obtain write permission.

There's also a subclass for secure connections :

class imaplib.**IMAP4_SSL**(*host=''*, *port=IMAP4_SSL_PORT*, ***, *ssl_context=None*, *timeout=None*)

This is a subclass derived from *IMAP4* that connects over an SSL encrypted socket (to use this class you need a socket module that was compiled with SSL support). If *host* is not specified, `''` (the local host) is used. If *port* is omitted, the standard IMAP4-over-SSL port (993) is used. *ssl_context* is a *ssl.SSLContext* object which allows bundling SSL configuration options, certificates and private keys into a single (potentially long-lived) structure. Please read *Security considerations* for best practices.

The optional *timeout* parameter specifies a timeout in seconds for the connection attempt. If timeout is not given or is `None`, the global default socket timeout is used.

Modifié dans la version 3.3 : *ssl_context* parameter was added.

Modifié dans la version 3.4 : The class now supports hostname check with *ssl.SSLContext.check_hostname* and *Server Name Indication* (see *ssl.HAS_SNI*).

Modifié dans la version 3.9 : The optional *timeout* parameter was added.

Modifié dans la version 3.12 : The deprecated *keyfile* and *certfile* parameters have been removed.

The second subclass allows for connections created by a child process :

class imaplib.**IMAP4_stream**(*command*)

This is a subclass derived from *IMAP4* that connects to the `stdin/stdout` file descriptors created by passing *command* to `subprocess.Popen()`.

The following utility functions are defined :

imaplib.**Internaldate2tuple**(*datestr*)

Parse an IMAP4 `INTERNALDATE` string and return corresponding local time. The return value is a *time.struct_time* tuple or `None` if the string has wrong format.

imaplib.**Int2AP**(*num*)

Converts an integer into a bytes representation using characters from the set [`A` .. `P`].

imaplib.**ParseFlags**(*flagstr*)

Converts an IMAP4 `FLAGS` response to a tuple of individual flags.

imaplib.**Time2Internaldate**(*date_time*)

Convert *date_time* to an IMAP4 `INTERNALDATE` representation. The return value is a string in the form : `"DD-Mmm-YYYY HH:MM:SS +HHMM"` (including double-quotes). The *date_time* argument can be a number (int or float) representing seconds since epoch (as returned by *time.time()*), a 9-tuple representing local time an instance of *time.struct_time* (as returned by *time.localtime()*), an aware instance of *datetime.datetime*, or a double-quoted string. In the last case, it is assumed to already be in the correct format.

Note that IMAP4 message numbers change as the mailbox changes ; in particular, after an `EXPUNGE` command performs deletions the remaining messages are renumbered. So it is highly advisable to use UIDs instead, with the UID command.

At the end of the module, there is a test section that contains a more extensive example of usage.

Voir aussi

Documents describing the protocol, sources for servers implementing it, by the University of Washington's IMAP Information Center can all be found at (**Source Code**) https://github.com/uw-imap/imap (**Not Maintained**).

22.13.1 IMAP4 Objects

All IMAP4rev1 commands are represented by methods of the same name, either uppercase or lowercase.

All arguments to commands are converted to strings, except for `AUTHENTICATE`, and the last argument to `APPEND` which is passed as an IMAP4 literal. If necessary (the string contains IMAP4 protocol-sensitive characters and isn't enclosed with either parentheses or double quotes) each string is quoted. However, the *password* argument to the `LOGIN` command is always quoted. If you want to avoid having an argument string quoted (eg : the *flags* argument to `STORE`) then enclose the string in parentheses (eg : `r'(\Deleted)'`).

Most commands return a tuple : `(type, [data, ...])` where *type* is usually `'OK'` or `'NO'`, and *data* is either the text from the command response, or mandated results from the command. Each *data* is either a `bytes`, or a tuple. If a tuple, then the first part is the header of the response, and the second part contains the data (ie : 'literal' value).

The *message_set* options to commands below is a string specifying one or more messages to be acted upon. It may be a simple message number (`'1'`), a range of message numbers (`'2:4'`), or a group of non-contiguous ranges separated by commas (`'1:3,6:9'`). A range can contain an asterisk to indicate an infinite upper bound (`'3:*'`).

An `IMAP4` instance has the following methods :

`IMAP4.`**`append`**(*mailbox*, *flags*, *date_time*, *message*)

Append *message* to named mailbox.

`IMAP4.`**`authenticate`**(*mechanism*, *authobject*)

Authenticate command --- requires response processing.

mechanism specifies which authentication mechanism is to be used - it should appear in the instance variable `capabilities` in the form `AUTH=mechanism`.

authobject must be a callable object :

```
data = authobject(response)
```

It will be called to process server continuation responses; the *response* argument it is passed will be `bytes`. It should return `bytes` *data* that will be base64 encoded and sent to the server. It should return `None` if the client abort response `*` should be sent instead.

Modifié dans la version 3.5 : string usernames and passwords are now encoded to `utf-8` instead of being limited to ASCII.

`IMAP4.`**`check`**()

Checkpoint mailbox on server.

`IMAP4.`**`close`**()

Close currently selected mailbox. Deleted messages are removed from writable mailbox. This is the recommended command before `LOGOUT`.

`IMAP4.`**`copy`**(*message_set*, *new_mailbox*)

Copy *message_set* messages onto end of *new_mailbox*.

`IMAP4.`**`create`**(*mailbox*)

Create new mailbox named *mailbox*.

`IMAP4.`**`delete`**(*mailbox*)

Delete old mailbox named *mailbox*.

`IMAP4.`**`deleteacl`**(*mailbox*, *who*)

Delete the ACLs (remove any rights) set for who on mailbox.

`IMAP4.`**`enable`**(*capability*)

Enable *capability* (see **RFC 5161**). Most capabilities do not need to be enabled. Currently only the `UTF8=ACCEPT` capability is supported (see **RFC 6855**).

Ajouté dans la version 3.5 : The `enable()` method itself, and **RFC 6855** support.

IMAP4.**expunge**()

Permanently remove deleted items from selected mailbox. Generates an EXPUNGE response for each deleted message. Returned data contains a list of EXPUNGE message numbers in order received.

IMAP4.**fetch**(*message_set*, *message_parts*)

Fetch (parts of) messages. *message_parts* should be a string of message part names enclosed within parentheses, eg : "(UID BODY[TEXT])". Returned data are tuples of message part envelope and data.

IMAP4.**getacl**(*mailbox*)

Get the ACLs for *mailbox*. The method is non-standard, but is supported by the Cyrus server.

IMAP4.**getannotation**(*mailbox*, *entry*, *attribute*)

Retrieve the specified ANNOTATIONs for *mailbox*. The method is non-standard, but is supported by the Cyrus server.

IMAP4.**getquota**(*root*)

Get the quota *root*'s resource usage and limits. This method is part of the IMAP4 QUOTA extension defined in rfc2087.

IMAP4.**getquotaroot**(*mailbox*)

Get the list of quota roots for the named *mailbox*. This method is part of the IMAP4 QUOTA extension defined in rfc2087.

IMAP4.**idle**(*duration=None*)

Return an Idler : an iterable context manager implementing the IMAP4 IDLE command as defined in **RFC 2177**.

The returned object sends the IDLE command when activated by the with statement, produces IMAP untagged responses via the *iterator* protocol, and sends DONE upon context exit.

All untagged responses that arrive after sending the IDLE command (including any that arrive before the server acknowledges the command) will be available via iteration. Any leftover responses (those not iterated in the with context) can be retrieved in the usual way after IDLE ends, using *IMAP4.response()*.

Responses are represented as (type, [data, ...]) tuples, as described in *IMAP4 Objects*.

The *duration* argument sets a maximum duration (in seconds) to keep idling, after which any ongoing iteration will stop. It can be an *int* or *float*, or None for no time limit. Callers wishing to avoid inactivity timeouts on servers that impose them should keep this at most 29 minutes (1740 seconds). Requires a socket connection ; *duration* must be None on *IMAP4_stream* connections.

```
>>> with M.idle(duration=29 * 60) as idler:
...     for typ, data in idler:
...         print(typ, data)
...
EXISTS [b'1']
RECENT [b'1']
```

Idler.**burst**(*interval=0.1*)

Yield a burst of responses no more than *interval* seconds apart (expressed as an *int* or *float*).

This *generator* is an alternative to iterating one response at a time, intended to aid in efficient batch processing. It retrieves the next response along with any immediately available subsequent responses. (For example, a rapid series of EXPUNGE responses after a bulk delete.)

Requires a socket connection ; does not work on *IMAP4_stream* connections.

```
>>> with M.idle() as idler:
...     # get a response and any others following by < 0.1 seconds
...     batch = list(idler.burst())
...     print(f'processing {len(batch)} responses...')
...     print(batch)
```

(suite sur la page suivante)

(suite de la page précédente)

```
...
processing 3 responses...
[('EXPUNGE', [b'2']), ('EXPUNGE', [b'1']), ('RECENT', [b'0'])]
```

Astuce

The `IDLE` context's maximum duration, as passed to `IMAP4.idle()`, is respected when waiting for the first response in a burst. Therefore, an expired `Idler` will cause this generator to return immediately without producing anything. Callers should consider this if using it in a loop.

Note

The iterator returned by `IMAP4.idle()` is usable only within a `with` statement. Before or after that context, unsolicited responses are collected internally whenever a command finishes, and can be retrieved with `IMAP4.response()`.

Note

The `Idler` class name and structure are internal interfaces, subject to change. Calling code can rely on its context management, iteration, and public method to remain stable, but should not subclass, instantiate, compare, or otherwise directly reference the class.

Ajouté dans la version 3.14.

`IMAP4.`**`list`**`(`[*directory*[, *pattern*]]`)`

List mailbox names in *directory* matching *pattern*. *directory* defaults to the top-level mail folder, and *pattern* defaults to match anything. Returned data contains a list of `LIST` responses.

`IMAP4.`**`login`**`(`*user*, *password*`)`

Identify the client using a plaintext password. The *password* will be quoted.

`IMAP4.`**`login_cram_md5`**`(`*user*, *password*`)`

Force use of `CRAM-MD5` authentication when identifying the client to protect the password. Will only work if the server `CAPABILITY` response includes the phrase `AUTH=CRAM-MD5`.

`IMAP4.`**`logout`**`()`

Shutdown connection to server. Returns server `BYE` response.

Modifié dans la version 3.8 : The method no longer ignores silently arbitrary exceptions.

`IMAP4.`**`lsub`**`(`*directory*='"", *pattern*='*'`)`

List subscribed mailbox names in directory matching pattern. *directory* defaults to the top level directory and *pattern* defaults to match any mailbox. Returned data are tuples of message part envelope and data.

`IMAP4.`**`myrights`**`(`*mailbox*`)`

Show my ACLs for a mailbox (i.e. the rights that I have on mailbox).

`IMAP4.`**`namespace`**`()`

Returns IMAP namespaces as defined in **RFC 2342**.

`IMAP4.`**`noop`**`()`

Send `NOOP` to server.

`IMAP4.`**`open`**`(`*host*, *port*, *timeout=None*`)`

Opens socket to *port* at *host*. The optional *timeout* parameter specifies a timeout in seconds for the connection

attempt. If timeout is not given or is `None`, the global default socket timeout is used. Also note that if the *timeout* parameter is set to be zero, it will raise a `ValueError` to reject creating a non-blocking socket. This method is implicitly called by the `IMAP4` constructor. The connection objects established by this method will be used in the `IMAP4.read()`, `IMAP4.readline()`, `IMAP4.send()`, and `IMAP4.shutdown()` methods. You may override this method.

Raises an *auditing event* `imaplib.open` with arguments `self`, `host`, `port`.

Modifié dans la version 3.9 : The *timeout* parameter was added.

`IMAP4.`**`partial`**(*message_num*, *message_part*, *start*, *length*)

Fetch truncated part of a message. Returned data is a tuple of message part envelope and data.

`IMAP4.`**`proxyauth`**(*user*)

Assume authentication as *user*. Allows an authorised administrator to proxy into any user's mailbox.

`IMAP4.`**`read`**(*size*)

Reads *size* bytes from the remote server. You may override this method.

`IMAP4.`**`readline`**()

Reads one line from the remote server. You may override this method.

`IMAP4.`**`recent`**()

Prompt server for an update. Returned data is `None` if no new messages, else value of `RECENT` response.

`IMAP4.`**`rename`**(*oldmailbox*, *newmailbox*)

Rename mailbox named *oldmailbox* to *newmailbox*.

`IMAP4.`**`response`**(*code*)

Return data for response *code* if received, or `None`. Returns the given code, instead of the usual type.

`IMAP4.`**`search`**(*charset*, *criterion*[, ...])

Search mailbox for matching messages. *charset* may be `None`, in which case no `CHARSET` will be specified in the request to the server. The IMAP protocol requires that at least one criterion be specified ; an exception will be raised when the server returns an error. *charset* must be `None` if the `UTF8=ACCEPT` capability was enabled using the `enable()` command.

Exemple :

```
# M is a connected IMAP4 instance...
typ, msgnums = M.search(None, 'FROM', '"LDJ"')

# or:
typ, msgnums = M.search(None, '(FROM "LDJ")')
```

`IMAP4.`**`select`**(*mailbox='INBOX'*, *readonly=False*)

Select a mailbox. Returned data is the count of messages in *mailbox* (`EXISTS` response). The default *mailbox* is `'INBOX'`. If the *readonly* flag is set, modifications to the mailbox are not allowed.

`IMAP4.`**`send`**(*data*)

Sends `data` to the remote server. You may override this method.

Raises an *auditing event* `imaplib.send` with arguments `self`, `data`.

`IMAP4.`**`setacl`**(*mailbox*, *who*, *what*)

Set an `ACL` for *mailbox*. The method is non-standard, but is supported by the `Cyrus` server.

`IMAP4.`**`setannotation`**(*mailbox*, *entry*, *attribute*[, ...])

Set `ANNOTATION`s for *mailbox*. The method is non-standard, but is supported by the `Cyrus` server.

`IMAP4.`**`setquota`**(*root*, *limits*)

Set the `quota` *root*'s resource *limits*. This method is part of the IMAP4 QUOTA extension defined in rfc2087.

IMAP4.**shutdown**()

Close connection established in `open`. This method is implicitly called by `IMAP4.logout()`. You may override this method.

IMAP4.**socket**()

Returns socket instance used to connect to server.

IMAP4.**sort**(*sort_criteria*, *charset*, *search_criterion*[, ...])

The `sort` command is a variant of `search` with sorting semantics for the results. Returned data contains a space separated list of matching message numbers.

Sort has two arguments before the *search_criterion* argument(s); a parenthesized list of *sort_criteria*, and the searching *charset*. Note that unlike `search`, the searching *charset* argument is mandatory. There is also a `uid sort` command which corresponds to `sort` the way that `uid search` corresponds to `search`. The `sort` command first searches the mailbox for messages that match the given searching criteria using the charset argument for the interpretation of strings in the searching criteria. It then returns the numbers of matching messages.

This is an `IMAP4rev1` extension command.

IMAP4.**starttls**(*ssl_context=None*)

Send a `STARTTLS` command. The *ssl_context* argument is optional and should be a `ssl.SSLContext` object. This will enable encryption on the IMAP connection. Please read *Security considerations* for best practices.

Ajouté dans la version 3.2.

Modifié dans la version 3.4 : The method now supports hostname check with `ssl.SSLContext.check_hostname` and *Server Name Indication* (see `ssl.HAS_SNI`).

IMAP4.**status**(*mailbox*, *names*)

Request named status conditions for *mailbox*.

IMAP4.**store**(*message_set*, *command*, *flag_list*)

Alters flag dispositions for messages in mailbox. *command* is specified by section 6.4.6 of **RFC 2060** as being one of "FLAGS", "+FLAGS", or "-FLAGS", optionally with a suffix of ".SILENT".

For example, to set the delete flag on all messages :

```
typ, data = M.search(None, 'ALL')
for num in data[0].split():
   M.store(num, '+FLAGS', '\\Deleted')
M.expunge()
```

Note

Creating flags containing ']' (for example : "[test]") violates **RFC 3501** (the IMAP protocol). However, imaplib has historically allowed creation of such tags, and popular IMAP servers, such as Gmail, accept and produce such flags. There are non-Python programs which also create such tags. Although it is an RFC violation and IMAP clients and servers are supposed to be strict, imaplib still continues to allow such tags to be created for backward compatibility reasons, and as of Python 3.6, handles them if they are sent from the server, since this improves real-world compatibility.

IMAP4.**subscribe**(*mailbox*)

Subscribe to new mailbox.

IMAP4.**thread**(*threading_algorithm*, *charset*, *search_criterion*[, ...])

The `thread` command is a variant of `search` with threading semantics for the results. Returned data contains a space separated list of thread members.

Thread members consist of zero or more messages numbers, delimited by spaces, indicating successive parent and child.

Thread has two arguments before the *search_criterion* argument(s) ; a *threading_algorithm*, and the searching *charset*. Note that unlike `search`, the searching *charset* argument is mandatory. There is also a `uid thread` command which corresponds to `thread` the way that `uid search` corresponds to `search`. The `thread` command first searches the mailbox for messages that match the given searching criteria using the *charset* argument for the interpretation of strings in the searching criteria. It then returns the matching messages threaded according to the specified threading algorithm.

This is an `IMAP4rev1` extension command.

`IMAP4.`**`uid`**(*command*, *arg*[, ...])

Execute command args with messages identified by UID, rather than message number. Returns response appropriate to command. At least one argument must be supplied ; if none are provided, the server will return an error and an exception will be raised.

`IMAP4.`**`unsubscribe`**(*mailbox*)

Unsubscribe from old mailbox.

`IMAP4.`**`unselect`**()

`imaplib.IMAP4.unselect()` frees server's resources associated with the selected mailbox and returns the server to the authenticated state. This command performs the same actions as *`imaplib.IMAP4.close()`*, except that no messages are permanently removed from the currently selected mailbox.

Ajouté dans la version 3.9.

`IMAP4.`**`xatom`**(*name*[, ...])

Allow simple extension commands notified by server in `CAPABILITY` response.

The following attributes are defined on instances of *`IMAP4`* :

`IMAP4.`**`PROTOCOL_VERSION`**

The most recent supported protocol in the `CAPABILITY` response from the server.

`IMAP4.`**`debug`**

Integer value to control debugging output. The initialize value is taken from the module variable `Debug`. Values greater than three trace each command.

`IMAP4.`**`utf8_enabled`**

Boolean value that is normally `False`, but is set to `True` if an *`enable()`* command is successfully issued for the `UTF8=ACCEPT` capability.

Ajouté dans la version 3.5.

22.13.2 IMAP4 Example

Here is a minimal example (without error checking) that opens a mailbox and retrieves and prints all messages :

```
import getpass, imaplib

M = imaplib.IMAP4(host='example.org')
M.login(getpass.getuser(), getpass.getpass())
M.select()
typ, data = M.search(None, 'ALL')
for num in data[0].split():
    typ, data = M.fetch(num, '(RFC822)')
    print('Message %s\n%s\n' % (num, data[0][1]))
M.close()
M.logout()
```

22.14 smtplib --- SMTP protocol client

Code source : Lib/smtplib.py

The *smtplib* module defines an SMTP client session object that can be used to send mail to any internet machine with an SMTP or ESMTP listener daemon. For details of SMTP and ESMTP operation, consult **RFC 821** (Simple Mail Transfer Protocol) and **RFC 1869** (SMTP Service Extensions).

Availability : not WASI.

This module does not work or is not available on WebAssembly. See *Plateformes WebAssembly* for more information.

class smtplib.**SMTP**(*host=''*, *port=0*, *local_hostname=None*, [*timeout*,]*source_address=None*)

An *SMTP* instance encapsulates an SMTP connection. It has methods that support a full repertoire of SMTP and ESMTP operations. If the optional *host* and *port* parameters are given, the SMTP *connect()* method is called with those parameters during initialization. If specified, *local_hostname* is used as the FQDN of the local host in the HELO/EHLO command. Otherwise, the local hostname is found using *socket.getfqdn()*. If the *connect()* call returns anything other than a success code, an *SMTPConnectError* is raised. The optional *timeout* parameter specifies a timeout in seconds for blocking operations like the connection attempt (if not specified, the global default timeout setting will be used). If the timeout expires, *TimeoutError* is raised. The optional *source_address* parameter allows binding to some specific source address in a machine with multiple network interfaces, and/or to some specific source TCP port. It takes a 2-tuple `(host, port)`, for the socket to bind to as its source address before connecting. If omitted (or if *host* or *port* are `''` and/or `0` respectively) the OS default behavior will be used.

For normal use, you should only require the initialization/connect, *sendmail()*, and *SMTP.quit()* methods. An example is included below.

The *SMTP* class supports the `with` statement. When used like this, the SMTP `QUIT` command is issued automatically when the `with` statement exits. E.g. :

```
>>> from smtplib import SMTP
>>> with SMTP("domain.org") as smtp:
...     smtp.noop()
...
(250, b'Ok')
>>>
```

All commands will raise an *auditing event* `smtplib.SMTP.send` with arguments `self` and `data`, where `data` is the bytes about to be sent to the remote host.

Modifié dans la version 3.3 : La prise en charge de l'instruction `with` a été ajoutée.

Modifié dans la version 3.3 : *source_address* argument was added.

Ajouté dans la version 3.5 : The SMTPUTF8 extension (**RFC 6531**) is now supported.

Modifié dans la version 3.9 : If the *timeout* parameter is set to be zero, it will raise a *ValueError* to prevent the creation of a non-blocking socket.

class smtplib.**SMTP_SSL**(*host=''*, *port=0*, *local_hostname=None*, *, [*timeout*,]*context=None*, *source_address=None*)

An *SMTP_SSL* instance behaves exactly the same as instances of *SMTP*. *SMTP_SSL* should be used for situations where SSL is required from the beginning of the connection and using `starttls()` is not appropriate. If *host* is not specified, the local host is used. If *port* is zero, the standard SMTP-over-SSL port (465) is used. The optional arguments *local_hostname*, *timeout* and *source_address* have the same meaning as they do in the *SMTP* class. *context*, also optional, can contain a *SSLContext* and allows configuring various aspects of the secure connection. Please read *Security considerations* for best practices.

Modifié dans la version 3.3 : *context* was added.

Modifié dans la version 3.3 : The *source_address* argument was added.

Modifié dans la version 3.4 : The class now supports hostname check with `ssl.SSLContext.check_hostname` and *Server Name Indication* (see `ssl.HAS_SNI`).

Modifié dans la version 3.9 : If the *timeout* parameter is set to be zero, it will raise a `ValueError` to prevent the creation of a non-blocking socket

Modifié dans la version 3.12 : The deprecated *keyfile* and *certfile* parameters have been removed.

class smtplib.**LMTP**(*host=''*, *port=LMTP_PORT*, *local_hostname=None*, *source_address=None*[, *timeout*])

The LMTP protocol, which is very similar to ESMTP, is heavily based on the standard SMTP client. It's common to use Unix sockets for LMTP, so our `connect()` method must support that as well as a regular host :port server. The optional arguments *local_hostname* and *source_address* have the same meaning as they do in the `SMTP` class. To specify a Unix socket, you must use an absolute path for *host*, starting with a '/'.

Authentication is supported, using the regular SMTP mechanism. When using a Unix socket, LMTP generally don't support or require any authentication, but your mileage might vary.

Modifié dans la version 3.9 : The optional *timeout* parameter was added.

A nice selection of exceptions is defined as well :

exception smtplib.**SMTPException**

Subclass of `OSError` that is the base exception class for all the other exceptions provided by this module.

Modifié dans la version 3.4 : SMTPException became subclass of `OSError`

exception smtplib.**SMTPServerDisconnected**

This exception is raised when the server unexpectedly disconnects, or when an attempt is made to use the `SMTP` instance before connecting it to a server.

exception smtplib.**SMTPResponseException**

Base class for all exceptions that include an SMTP error code. These exceptions are generated in some instances when the SMTP server returns an error code. The error code is stored in the `smtp_code` attribute of the error, and the `smtp_error` attribute is set to the error message.

exception smtplib.**SMTPSenderRefused**

Sender address refused. In addition to the attributes set by on all `SMTPResponseException` exceptions, this sets 'sender' to the string that the SMTP server refused.

exception smtplib.**SMTPRecipientsRefused**

All recipient addresses refused. The errors for each recipient are accessible through the attribute `recipients`, which is a dictionary of exactly the same sort as `SMTP.sendmail()` returns.

exception smtplib.**SMTPDataError**

The SMTP server refused to accept the message data.

exception smtplib.**SMTPConnectError**

Error occurred during establishment of a connection with the server.

exception smtplib.**SMTPHeloError**

The server refused our `HELO` message.

exception smtplib.**SMTPNotSupportedError**

The command or option attempted is not supported by the server.

Ajouté dans la version 3.5.

exception smtplib.**SMTPAuthenticationError**

SMTP authentication went wrong. Most probably the server didn't accept the username/password combination provided.

Voir aussi

RFC 821 - Simple Mail Transfer Protocol
Protocol definition for SMTP. This document covers the model, operating procedure, and protocol details for SMTP.

RFC 1869 - SMTP Service Extensions
Definition of the ESMTP extensions for SMTP. This describes a framework for extending SMTP with new commands, supporting dynamic discovery of the commands provided by the server, and defines a few additional commands.

22.14.1 SMTP Objects

An `SMTP` instance has the following methods :

`SMTP.set_debuglevel(level)`

Set the debug output level. A value of 1 or `True` for *level* results in debug messages for connection and for all messages sent to and received from the server. A value of 2 for *level* results in these messages being timestamped.

Modifié dans la version 3.5 : Added debuglevel 2.

`SMTP.docmd(cmd, args='')`

Send a command *cmd* to the server. The optional argument *args* is simply concatenated to the command, separated by a space.

This returns a 2-tuple composed of a numeric response code and the actual response line (multiline responses are joined into one long line.)

In normal operation it should not be necessary to call this method explicitly. It is used to implement other methods and may be useful for testing private extensions.

If the connection to the server is lost while waiting for the reply, `SMTPServerDisconnected` will be raised.

`SMTP.connect(host='localhost', port=0)`

Connect to a host on a given port. The defaults are to connect to the local host at the standard SMTP port (25). If the hostname ends with a colon (`':'`) followed by a number, that suffix will be stripped off and the number interpreted as the port number to use. This method is automatically invoked by the constructor if a host is specified during instantiation. Returns a 2-tuple of the response code and message sent by the server in its connection response.

Raises an *auditing event* `smtplib.connect` with arguments `self`, `host`, `port`.

`SMTP.helo(name='')`

Identify yourself to the SMTP server using `HELO`. The hostname argument defaults to the fully qualified domain name of the local host. The message returned by the server is stored as the `helo_resp` attribute of the object.

In normal operation it should not be necessary to call this method explicitly. It will be implicitly called by the `sendmail()` when necessary.

`SMTP.ehlo(name='')`

Identify yourself to an ESMTP server using `EHLO`. The hostname argument defaults to the fully qualified domain name of the local host. Examine the response for ESMTP option and store them for use by `has_extn()`. Also sets several informational attributes : the message returned by the server is stored as the `ehlo_resp` attribute, `does_esmtp` is set to `True` or `False` depending on whether the server supports ESMTP, and `esmtp_features` will be a dictionary containing the names of the SMTP service extensions this server supports, and their parameters (if any).

Unless you wish to use `has_extn()` before sending mail, it should not be necessary to call this method explicitly. It will be implicitly called by `sendmail()` when necessary.

SMTP.**ehlo_or_helo_if_needed**()

This method calls `ehlo()` and/or `helo()` if there has been no previous EHLO or HELO command this session. It tries ESMTP EHLO first.

SMTPHeloError

The server didn't reply properly to the HELO greeting.

SMTP.**has_extn**(*name*)

Return `True` if *name* is in the set of SMTP service extensions returned by the server, `False` otherwise. Case is ignored.

SMTP.**verify**(*address*)

Check the validity of an address on this server using SMTP VRFY. Returns a tuple consisting of code 250 and a full **RFC 822** address (including human name) if the user address is valid. Otherwise returns an SMTP error code of 400 or greater and an error string.

> **Note**
>
> Many sites disable SMTP VRFY in order to foil spammers.

SMTP.**login**(*user*, *password*, *, *initial_response_ok=True*)

Log in on an SMTP server that requires authentication. The arguments are the username and the password to authenticate with. If there has been no previous EHLO or HELO command this session, this method tries ESMTP EHLO first. This method will return normally if the authentication was successful, or may raise the following exceptions :

SMTPHeloError

The server didn't reply properly to the HELO greeting.

SMTPAuthenticationError

The server didn't accept the username/password combination.

SMTPNotSupportedError

The AUTH command is not supported by the server.

SMTPException

No suitable authentication method was found.

Each of the authentication methods supported by `smtplib` are tried in turn if they are advertised as supported by the server. See `auth()` for a list of supported authentication methods. *initial_response_ok* is passed through to `auth()`.

Optional keyword argument *initial_response_ok* specifies whether, for authentication methods that support it, an "initial response" as specified in **RFC 4954** can be sent along with the AUTH command, rather than requiring a challenge/response.

Modifié dans la version 3.5 : `SMTPNotSupportedError` may be raised, and the *initial_response_ok* parameter was added.

SMTP.**auth**(*mechanism*, *authobject*, *, *initial_response_ok=True*)

Issue an SMTP AUTH command for the specified authentication *mechanism*, and handle the challenge response via *authobject*.

mechanism specifies which authentication mechanism is to be used as argument to the AUTH command; the valid values are those listed in the `auth` element of `esmtp_features`.

authobject must be a callable object taking an optional single argument :

```
data = authobject(challenge=None)
```

If optional keyword argument *initial_response_ok* is true, `authobject()` will be called first with no argument. It can return the **RFC 4954** "initial response" ASCII `str` which will be encoded and sent with the AUTH command as below. If the `authobject()` does not support an initial response (e.g. because it requires a challenge), it should return `None` when called with `challenge=None`. If *initial_response_ok* is false, then `authobject()` will not be called first with `None`.

If the initial response check returns `None`, or if *initial_response_ok* is false, `authobject()` will be called to process the server's challenge response ; the *challenge* argument it is passed will be a `bytes`. It should return ASCII `str` *data* that will be base64 encoded and sent to the server.

The `SMTP` class provides `authobjects` for the `CRAM-MD5`, `PLAIN`, and `LOGIN` mechanisms ; they are named `SMTP.auth_cram_md5`, `SMTP.auth_plain`, and `SMTP.auth_login` respectively. They all require that the `user` and `password` properties of the `SMTP` instance are set to appropriate values.

User code does not normally need to call `auth` directly, but can instead call the `login()` method, which will try each of the above mechanisms in turn, in the order listed. `auth` is exposed to facilitate the implementation of authentication methods not (or not yet) supported directly by `smtplib`.

Ajouté dans la version 3.5.

`SMTP.`**`starttls`**(*, *context=None*)

Put the SMTP connection in TLS (Transport Layer Security) mode. All SMTP commands that follow will be encrypted. You should then call `ehlo()` again.

If *keyfile* and *certfile* are provided, they are used to create an `ssl.SSLContext`.

Optional *context* parameter is an `ssl.SSLContext` object ; This is an alternative to using a keyfile and a certfile and if specified both *keyfile* and *certfile* should be `None`.

If there has been no previous `EHLO` or `HELO` command this session, this method tries ESMTP `EHLO` first.

Modifié dans la version 3.12 : The deprecated *keyfile* and *certfile* parameters have been removed.

SMTPHeloError

The server didn't reply properly to the `HELO` greeting.

SMTPNotSupportedError

The server does not support the STARTTLS extension.

RuntimeError

SSL/TLS support is not available to your Python interpreter.

Modifié dans la version 3.3 : *context* was added.

Modifié dans la version 3.4 : The method now supports hostname check with `SSLContext.check_hostname` and *Server Name Indicator* (see `HAS_SNI`).

Modifié dans la version 3.5 : The error raised for lack of STARTTLS support is now the `SMTPNotSupportedError` subclass instead of the base `SMTPException`.

`SMTP.`**`sendmail`**(*from_addr*, *to_addrs*, *msg*, *mail_options=()*, *rcpt_options=()*)

Send mail. The required arguments are an **RFC 822** from-address string, a list of **RFC 822** to-address strings (a bare string will be treated as a list with 1 address), and a message string. The caller may pass a list of ESMTP options (such as `8bitmime`) to be used in `MAIL FROM` commands as *mail_options*. ESMTP options (such as `DSN` commands) that should be used with all `RCPT` commands can be passed as *rcpt_options*. (If you need to use different ESMTP options to different recipients you have to use the low-level methods such as `mail()`, `rcpt()` and `data()` to send the message.)

> **Note**
>
> The *from_addr* and *to_addrs* parameters are used to construct the message envelope used by the transport agents. `sendmail` does not modify the message headers in any way.

msg may be a string containing characters in the ASCII range, or a byte string. A string is encoded to bytes using the ascii codec, and lone `\r` and `\n` characters are converted to `\r\n` characters. A byte string is not modified.

If there has been no previous `EHLO` or `HELO` command this session, this method tries ESMTP `EHLO` first. If the server does ESMTP, message size and each of the specified options will be passed to it (if the option is in the feature set the server advertises). If `EHLO` fails, `HELO` will be tried and ESMTP options suppressed.

This method will return normally if the mail is accepted for at least one recipient. Otherwise it will raise an exception. That is, if this method does not raise an exception, then someone should get your mail. If this method

does not raise an exception, it returns a dictionary, with one entry for each recipient that was refused. Each entry contains a tuple of the SMTP error code and the accompanying error message sent by the server.

If `SMTPUTF8` is included in *mail_options*, and the server supports it, *from_addr* and *to_addrs* may contain non-ASCII characters.

This method may raise the following exceptions :

SMTPRecipientsRefused
All recipients were refused. Nobody got the mail. The `recipients` attribute of the exception object is a dictionary with information about the refused recipients (like the one returned when at least one recipient was accepted).

SMTPHeloError
The server didn't reply properly to the `HELO` greeting.

SMTPSenderRefused
The server didn't accept the *from_addr*.

SMTPDataError
The server replied with an unexpected error code (other than a refusal of a recipient).

SMTPNotSupportedError
`SMTPUTF8` was given in the *mail_options* but is not supported by the server.

Unless otherwise noted, the connection will be open even after an exception is raised.

Modifié dans la version 3.2 : *msg* may be a byte string.

Modifié dans la version 3.5 : `SMTPUTF8` support added, and `SMTPNotSupportedError` may be raised if `SMTPUTF8` is specified but the server does not support it.

`SMTP.`**`send_message`**(*msg*, *from_addr=None*, *to_addrs=None*, *mail_options=()*, *rcpt_options=()*)

This is a convenience method for calling `sendmail()` with the message represented by an `email.message.Message` object. The arguments have the same meaning as for `sendmail()`, except that *msg* is a `Message` object.

If *from_addr* is `None` or *to_addrs* is `None`, `send_message` fills those arguments with addresses extracted from the headers of *msg* as specified in **RFC 5322** : *from_addr* is set to the `Sender` field if it is present, and otherwise to the `From` field. *to_addrs* combines the values (if any) of the `To`, `Cc`, and `Bcc` fields from *msg*. If exactly one set of `Resent-*` headers appear in the message, the regular headers are ignored and the `Resent-*` headers are used instead. If the message contains more than one set of `Resent-*` headers, a `ValueError` is raised, since there is no way to unambiguously detect the most recent set of `Resent-` headers.

`send_message` serializes *msg* using `BytesGenerator` with `\r\n` as the *linesep*, and calls `sendmail()` to transmit the resulting message. Regardless of the values of *from_addr* and *to_addrs*, `send_message` does not transmit any `Bcc` or `Resent-Bcc` headers that may appear in *msg*. If any of the addresses in *from_addr* and *to_addrs* contain non-ASCII characters and the server does not advertise `SMTPUTF8` support, an `SMTPNotSupportedError` is raised. Otherwise the `Message` is serialized with a clone of its `policy` with the `utf8` attribute set to `True`, and `SMTPUTF8` and `BODY=8BITMIME` are added to *mail_options*.

Ajouté dans la version 3.2.

Ajouté dans la version 3.5 : Support for internationalized addresses (`SMTPUTF8`).

`SMTP.`**`quit`**()

Terminate the SMTP session and close the connection. Return the result of the SMTP `QUIT` command.

Low-level methods corresponding to the standard SMTP/ESMTP commands `HELP`, `RSET`, `NOOP`, `MAIL`, `RCPT`, and `DATA` are also supported. Normally these do not need to be called directly, so they are not documented here. For details, consult the module code.

22.14.2 SMTP Example

This example prompts the user for addresses needed in the message envelope ('To' and 'From' addresses), and the message to be delivered. Note that the headers to be included with the message must be included in the message as entered ; this example doesn't do any processing of the **RFC 822** headers. In particular, the 'To' and 'From' addresses must be included in the message headers explicitly :

```
import smtplib

def prompt(title):
    return input(title).strip()

from_addr = prompt("From: ")
to_addrs  = prompt("To: ").split()
print("Enter message, end with ^D (Unix) or ^Z (Windows):")

# Add the From: and To: headers at the start!
lines = [f"From: {from_addr}", f"To: {', '.join(to_addrs)}", ""]
while True:
    try:
        line = input()
    except EOFError:
        break
    else:
        lines.append(line)

msg = "\r\n".join(lines)
print("Message length is", len(msg))

server = smtplib.SMTP("localhost")
server.set_debuglevel(1)
server.sendmail(from_addr, to_addrs, msg)
server.quit()
```

Note

In general, you will want to use the `email` package's features to construct an email message, which you can then send via `send_message()` ; see *email : Exemples*.

22.15 `uuid` --- UUID objects according to RFC 9562

Code source : Lib/uuid.py

This module provides immutable `UUID` objects (the `UUID` class) and the functions `uuid1()`, `uuid3()`, `uuid4()`, `uuid5()`, and `uuid.uuid8()` for generating version 1, 3, 4, 5, and 8 UUIDs as specified in **RFC 9562** (which supersedes **RFC 4122**).

Utilisez `uuid1()` ou `uuid4()` si votre but est de produire un identifiant unique. Notez que `uuid1()` peut dévoiler des informations personnelles car l'UUID produit contient l'adresse réseau de l'ordinateur. En revanche, `uuid4()` génère un UUID aléatoire.

Depending on support from the underlying platform, `uuid1()` may or may not return a "safe" UUID. A safe UUID is one which is generated using synchronization methods that ensure no two processes can obtain the same UUID. All instances of `UUID` have an `is_safe` attribute which relays any information about the UUID's safety, using this enumeration :

class uuid.**SafeUUID**

Ajouté dans la version 3.7.

safe

L'UUID a été généré par la plateforme en utilisant une méthode sûre dans un contexte de parallélisme par processus.

unsafe

L'UUID n'a pas été généré par une méthode sûre dans un contexte de parallélisme par processus.

unknown

La plateforme ne précise pas si l'UUID a été généré de façon sûre ou non.

class uuid.**UUID**(*hex=None*, *bytes=None*, *bytes_le=None*, *fields=None*, *int=None*, *version=None*, *, *is_safe=SafeUUID.unknown*)

Produit un UUID à partir soit d'une chaîne de 32 chiffres hexadécimaux, soit une chaîne de 16 octets gros-boutiste (argument *bytes*), soit une chaîne de 16 octets petit-boutiste (argument *bytes_le*), soit un sextuplet d'entiers (32-bit *time_low*, 16-bit *time_mid*, 16-bit *time_hi_version*, 8-bit *clock_seq_hi_variant*, 8-bit *clock_seq_low*, 48-bit *node*) (argument *fields*), soit un unique entier sur 128 bits (argument *int*). Lorsque la fonction reçoit une chaîne de chiffres hexadécimaux, les accolades, les tirets et le préfixe URN sont facultatifs. Par exemple, toutes les expressions ci-dessous génèrent le même UUID :

```
UUID('{12345678-1234-5678-1234-567812345678}')
UUID('12345678123456781234567812345678')
UUID('urn:uuid:12345678-1234-5678-1234-567812345678')
UUID(bytes=b'\x12\x34\x56\x78'*4)
UUID(bytes_le=b'\x78\x56\x34\x12\x34\x12\x78\x56' +
              b'\x12\x34\x56\x78\x12\x34\x56\x78')
UUID(fields=(0x12345678, 0x1234, 0x5678, 0x12, 0x34, 0x567812345678))
UUID(int=0x12345678123456781234567812345678)
```

Exactly one of *hex*, *bytes*, *bytes_le*, *fields*, or *int* must be given. The *version* argument is optional; if given, the resulting UUID will have its variant and version number set according to **RFC 9562**, overriding bits in the given *hex*, *bytes*, *bytes_le*, *fields*, or *int*.

La comparaison de deux objets UUID se fait en comparant leurs attributs `UUID.int`. La comparaison avec un objet autre qu'un UUID lève une exception `TypeError`.

`str(uuid)` renvoie une chaîne de caractères de la forme `12345678-1234-5678-1234-567812345678` représentant l'UUID par une chaîne de 32 chiffres hexadécimaux.

Les instances de `UUID` possèdent les attributs suivants en lecture seule :

UUID.**bytes**

L'UUID représenté comme une chaîne de 16 octets (contenant les six champs entiers dans l'ordre gros-boutiste).

UUID.**bytes_le**

L'UUID représenté comme une chaîne de 16 octets (avec *time_low*, *time_mid* et *time_hi_version* dans l'ordre petit-boutiste).

UUID.**fields**

Un sextuplet contenant les champs entiers de l'UUID, également accessibles en tant que six attributs individuels et deux attributs dérivés :

Champ	Signification
`UUID.time_low`	The first 32 bits of the UUID.
`UUID.time_mid`	The next 16 bits of the UUID.
`UUID.time_hi_version`	The next 16 bits of the UUID.
`UUID.clock_seq_hi_variant`	The next 8 bits of the UUID.
`UUID.clock_seq_low`	The next 8 bits of the UUID.
`UUID.node`	The last 48 bits of the UUID.
`UUID.time`	The 60-bit timestamp.
`UUID.clock_seq`	The 14-bit sequence number.

`UUID.hex`

The UUID as a 32-character lowercase hexadecimal string.

`UUID.int`

Représentation de l'UUId sous forme d'un entier de 128 bits.

`UUID.urn`

The UUID as a URN as specified in **RFC 9562**.

`UUID.variant`

Variante de l'UUID. Celle-ci détermine l'agencement interne de l'UUID. Les valeurs possibles sont les constantes suivantes : *RESERVED_NCS*, *RFC_4122*, *RESERVED_MICROSOFT* ou *RESERVED_FUTURE*.

`UUID.version`

The UUID version number (1 through 8, meaningful only when the variant is *RFC_4122*).

Modifié dans la version 3.14 : Added UUID version 8.

`UUID.is_safe`

Valeur de l'énumération *SafeUUID* indiquant si la plateforme a généré l'UUID d'une façon sûre dans un contexte de parallélisme par processus.

Ajouté dans la version 3.7.

Le module *uu* définit les fonctions suivantes :

`uuid.getnode()`

Renvoie l'adresse réseau matérielle sous forme d'un entier positif sur 48 bits. Cette fonction peut faire appel à un programme externe relativement lent lors de sa première exécution. Si toutes les tentatives d'obtenir l'adresse matérielle échouent, un nombre aléatoire sur 48 bit avec le bit de *multicast* (bit de poids faible du premier octet) à 1 est généré, comme recommandé par la **RFC 4122**. Ici, « adresse matérielle » correspond à l'adresse MAC d'une interface réseau. Sur une machine avec plusieurs interfaces réseau, les adresses MAC de type UUA (*universally administered address*, pour lesquelles le second bit de poids faible est à zéro) sont prioritaires par

rapport aux autres adresses MAC. Aucune autre garantie n'est donnée sur l'ordre dans lequel les interfaces sont choisies.

Modifié dans la version 3.7 : Les adresses MAC de type UUA sont préférées par rapport aux adresses locales car ces dernières ne sont pas nécessairement uniques.

`uuid.`**`uuid1`**(*node=None*, *clock_seq=None*)

Génère un UUID à partir d'un identifiant hôte, d'un numéro de séquence et de l'heure actuelle. Si *node* n'est pas spécifié, la fonction `getnode()` est appelée pour obtenir l'adresse matérielle. *clock_seq* est utilisé comme numéro de séquence s'il est spécifié, sinon un numéro aléatoire sur 14 bits est utilisé à la place.

`uuid.`**`uuid3`**(*namespace*, *name*)

Generate a UUID based on the MD5 hash of a namespace identifier (which is a UUID) and a name (which is a `bytes` object or a string that will be encoded using UTF-8).

`uuid.`**`uuid4`**()

Génère un UUID aléatoire.

`uuid.`**`uuid5`**(*namespace*, *name*)

Generate a UUID based on the SHA-1 hash of a namespace identifier (which is a UUID) and a name (which is a `bytes` object or a string that will be encoded using UTF-8).

`uuid.`**`uuid8`**(*a=None*, *b=None*, *c=None*)

Generate a pseudo-random UUID according to **RFC 9562, §5.8**.

When specified, the parameters *a*, *b* and *c* are expected to be positive integers of 48, 12 and 62 bits respectively. If they exceed their expected bit count, only their least significant bits are kept ; non-specified arguments are substituted for a pseudo-random integer of appropriate size.

Ajouté dans la version 3.14.

Le module `uuid` définit les identifiants d'espaces de noms suivants (pour `uuid3()` et `uuid5()`).

`uuid.`**`NAMESPACE_DNS`**

When this namespace is specified, the *name* string is a fully qualified domain name.

`uuid.`**`NAMESPACE_URL`**

Lorsque cet espace de nom est spécifié, la chaîne *name* doit être une URL.

`uuid.`**`NAMESPACE_OID`**

Lorsque cet espace de nom est spécifié, la chaîne *name* doit être un OID ISO.

`uuid.`**`NAMESPACE_X500`**

Lorsque cet espace de nom est spécifié, la chaîne *name* doit être un DN X.500 au format texte ou DER.

The `uuid` module defines the following constants for the possible values of the `variant` attribute :

`uuid.`**`RESERVED_NCS`**

Réservé pour la compatibilité NCS.

`uuid.`**`RFC_4122`**

Specifies the UUID layout given in **RFC 4122**. This constant is kept for backward compatibility even though **RFC 4122** has been superseded by **RFC 9562**.

`uuid.`**`RESERVED_MICROSOFT`**

Réservé pour la compatibilité Microsoft.

`uuid.`**`RESERVED_FUTURE`**

Réservé pour un usage futur.

The `uuid` module defines the special Nil and Max UUID values :

`uuid.NIL`

A special form of UUID that is specified to have all 128 bits set to zero according to **RFC 9562, §5.9**.

Ajouté dans la version 3.14.

`uuid.MAX`

A special form of UUID that is specified to have all 128 bits set to one according to **RFC 9562, §5.10**.

Ajouté dans la version 3.14.

Voir aussi

RFC 9562 - A Universally Unique IDentifier (UUID) URN Namespace
Cette spécification (en anglais) définit un espace de noms *Uniform Resource Name* pour les UUID, leur format interne et les méthodes permettant de les générer.

22.15.1 Utilisation en ligne de commande.

Ajouté dans la version 3.12.

The `uuid` module can be executed as a script from the command line.

```
python -m uuid [-h] [-u {uuid1,uuid3,uuid4,uuid5,uuid8}] [-n NAMESPACE] [-N NAME]
```

Les options suivantes sont acceptées :

-h, --help

Affiche un message d'aide et quitte.

-u <uuid>
--uuid <uuid>

Specify the function name to use to generate the uuid. By default `uuid4()` is used.

Ajouté dans la version 3.14 : Allow generating UUID version 8.

-n <namespace>
--namespace <namespace>

The namespace is a `UUID`, or `@ns` where `ns` is a well-known predefined UUID addressed by namespace name. Such as `@dns`, `@url`, `@oid`, and `@x500`. Only required for `uuid3()` / `uuid5()` functions.

-N <name>
--name <name>

The name used as part of generating the uuid. Only required for `uuid3()` / `uuid5()` functions.

22.15.2 Exemple

Voici quelques exemples classiques d'utilisation du module `uuid` :

```
>>> import uuid

>>> # make a UUID based on the host ID and current time
>>> uuid.uuid1()
UUID('a8098c1a-f86e-11da-bd1a-00112444be1e')

>>> # make a UUID using an MD5 hash of a namespace UUID and a name
>>> uuid.uuid3(uuid.NAMESPACE_DNS, 'python.org')
UUID('6fa459ea-ee8a-3ca4-894e-db77e160355e')

>>> # make a random UUID
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> uuid.uuid4()
UUID('16fd2706-8baf-433b-82eb-8c7fada847da')

>>> # make a UUID using a SHA-1 hash of a namespace UUID and a name
>>> uuid.uuid5(uuid.NAMESPACE_DNS, 'python.org')
UUID('886313e1-3b8a-5372-9b90-0c9aee199e5d')

>>> # make a UUID from a string of hex digits (braces and hyphens ignored)
>>> x = uuid.UUID('{00010203-0405-0607-0809-0a0b0c0d0e0f}')

>>> # convert a UUID to a string of hex digits in standard form
>>> str(x)
'00010203-0405-0607-0809-0a0b0c0d0e0f'

>>> # get the raw 16 bytes of the UUID
>>> x.bytes
b'\x00\x01\x02\x03\x04\x05\x06\x07\x08\t\n\x0b\x0c\r\x0e\x0f'

>>> # make a UUID from a 16-byte string
>>> uuid.UUID(bytes=x.bytes)
UUID('00010203-0405-0607-0809-0a0b0c0d0e0f')

>>> # get the Nil UUID
>>> uuid.NIL
UUID('00000000-0000-0000-0000-000000000000')

>>> # get the Max UUID
>>> uuid.MAX
UUID('ffffffff-ffff-ffff-ffff-ffffffffffff')
```

22.15.3 Command-Line Example

Here are some examples of typical usage of the `uuid` command line interface :

```
# generate a random uuid - by default uuid4() is used
$ python -m uuid

# generate a uuid using uuid1()
$ python -m uuid -u uuid1

# generate a uuid using uuid5
$ python -m uuid -u uuid5 -n @url -N example.com
```

22.16 `socketserver` --- A framework for network servers

Code source : Lib/socketserver.py

Le module `socketserver` permet de simplifier le développement de serveurs réseaux.

Availability : not WASI.

This module does not work or is not available on WebAssembly. See *Plateformes WebAssembly* for more information.

Il existe quatre classes concrètes fondamentales :

class socketserver.**TCPServer**(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)

Cette classe permet de créer des flux continus de données entre un client et un serveur en utilisant le protocole internet TCP. Si *bind_and_activate* est vrai, le constructeur tente automatiquement d'invoquer `server_bind()` et `server_activate()`. Les autres paramètres sont passés à la classe de base `BaseServer`.

class socketserver.**UDPServer**(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)

Cette classe utilise des datagrammes, qui sont des paquets d'information pouvant arriver dans le désordre, voire être perdus, durant le transport. Les paramètres sont identiques à `TCPServer`.

class socketserver.**UnixStreamServer**(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)

class socketserver.**UnixDatagramServer**(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)

Ces classes, moins fréquemment utilisées, sont similaires aux classes pour TCP et UDP mais utilisent des connecteurs UNIX ; ces connecteurs ne sont disponibles que sur les plateformes UNIX. Les paramètres sont identiques à `TCPServer`.

Ces quatre classes traitent les requêtes de façon *synchrone* : chaque requête doit être terminée avant de pouvoir traiter la suivante. Cette méthode n'est pas adaptée si les requêtes prennent beaucoup de temps à être traitées. Cela peut arriver si les requêtes demandent beaucoup de calcul, ou si elles renvoient beaucoup de données que le client peine à traiter. Dans ce cas, la solution est de créer un processus ou un fil d'exécution séparé pour chaque requête ; les classes de mélange `ForkingMixIn` et `ThreadingMixIn` peuvent être utilisées pour réaliser ce comportement asynchrone.

La création d'un serveur requiert plusieurs étapes. Premièrement, vous devez créer une classe pour gérer les requêtes en héritant de `BaseRequestHandler` et surcharger sa méthode `handle()`, laquelle traitera les requêtes entrantes. Deuxièmement, vous devez instancier l'une des classes serveurs et lui passer l'adresse du serveur ainsi que la classe gérant les requêtes. Il est recommandé de faire ceci dans une instruction `with`. Ensuite, appelez la méthode `handle_request()` ou `serve_forever()` de l'objet serveur afin de traiter une ou plusieurs requêtes. Enfin, appelez la méthode `server_close()` pour fermer le connecteur (à moins que vous n'ayez utilisé une instruction `with`).

Lorsque vous héritez de `ThreadingMixIn` pour déléguer les connexions à différent fils d'exécution, vous devez déclarer explicitement comment les fils d'exécution doivent se comporter en cas d'arrêt abrupt. La classe `ThreadingMixIn` définit un attribut *daemon_threads*, indiquant si le serveur doit attendre la fin des fils d'exécution ou non. Vous pouvez utiliser cet attribut si vous souhaitez que les fils d'exécution soient autonomes. La valeur par défaut est `False`, indiquant que Python ne doit pas quitter avant que tous les fils d'exécution créés par `ThreadingMixIn` ne soient terminés.

Toutes les classes de serveurs exposent les mêmes méthodes et attributs, peu importe le protocole réseau utilisé.

22.16.1 Notes sur la création de serveurs

Il y a cinq classes dans la hiérarchie. Quatre d'entre elles représentent des serveurs synchrones de quatre types différents :

```
+------------+
| BaseServer |
+------------+
      |
      v
+-----------+        +------------------+
| TCPServer |------->| UnixStreamServer |
+-----------+        +------------------+
      |
      v
+-----------+        +--------------------+
| UDPServer |------->| UnixDatagramServer |
+-----------+        +--------------------+
```

Note that *UnixDatagramServer* derives from *UDPServer*, not from *UnixStreamServer* --- the only difference between an IP and a Unix server is the address family.

class socketserver.**ForkingMixIn**

class socketserver.**ThreadingMixIn**

Des versions utilisant des fils d'exécution ou des processus peuvent être créées pour chaque type de serveur, en utilisant ces classes de mélange. Par exemple, *ThreadingUDPServer* est créé comme suit :

```
class ThreadingUDPServer(ThreadingMixIn, UDPServer):
    pass
```

La classe de mélange est en premier car elle surcharge une méthode définie dans *UDPServer*. Configurer les différents attributs changera également le comportement du serveur.

La classe *ForkingMixIn* et les classes créant des processus mentionnées ci-dessous sont uniquement disponibles sur les plateformes POSIX prenant en charge *fork()*.

block_on_close

ForkingMixIn.server_close waits until all child processes complete, except if *block_on_close* attribute is `False`.

ThreadingMixIn.server_close waits until all non-daemon threads complete, except if *block_on_close* attribute is `False`.

max_children

Specify how many child processes will exist to handle requests at a time for *ForkingMixIn*. If the limit is reached, new requests will wait until one child process has finished.

daemon_threads

For *ThreadingMixIn* use daemonic threads by setting *ThreadingMixIn.daemon_threads* to `True` to not wait until threads complete.

Modifié dans la version 3.7 : *ForkingMixIn.server_close* and *ThreadingMixIn.server_close* now waits until all child processes and non-daemonic threads complete. Add a new *ForkingMixIn.block_on_close* class attribute to opt-in for the pre-3.7 behaviour.

class socketserver.**ForkingTCPServer**

class socketserver.**ForkingUDPServer**

class socketserver.**ThreadingTCPServer**

class socketserver.**ThreadingUDPServer**

class socketserver.**ForkingUnixStreamServer**

class socketserver.**ForkingUnixDatagramServer**

class socketserver.**ThreadingUnixStreamServer**

class socketserver.**ThreadingUnixDatagramServer**

Ces classes sont prédéfinies en utilisant les classes de mélange.

Ajouté dans la version 3.12 : The `ForkingUnixStreamServer` and `ForkingUnixDatagramServer` classes were added.

Pour implémenter un service, vous devez créer une classe héritant de *BaseRequestHandler* et redéfinir sa méthode *handle()*. Ensuite, vous pourrez créer différentes versions de votre service en combinant les classes serveurs avec votre classe de gestion des requêtes. Cette classe de gestion des requêtes doit être différente pour les services utilisant des datagrammes ou des flux de données. Cette contrainte peut être dissimulée en utilisant les classes de gestion dérivées *StreamRequestHandler* ou *DatagramRequestHandler*.

Bien entendu, vous devrez toujours utiliser votre tête ! Par exemple, utiliser un serveur utilisant des processus clonés (*forking*) n'aurait aucun sens si le serveur garde en mémoire des états pouvant être modifiés par les requêtes reçues. En effet, un processus enfant traitant une requête n'aurait alors aucun moyen de propager le nouvel état à son parent. Dans ce cas, vous devez utiliser un serveur utilisant des fils d'exécution, mais cela demande probablement d'utiliser des verrous pour protéger l'intégrité des données partagées.

D'un autre côté, si vous développez un serveur HTTP qui a toutes ses données stockées hors de la mémoire (sur un système de fichiers par exemple), une classe synchrone rendrait le service sourd à toute nouvelle requête aussi longtemps qu'une précédente soit en cours de traitement. Cette situation pourrait perdurer pendant un long moment si le client prend du temps à recevoir toutes les données demandées. Dans ce cas, un serveur utilisant des processus ou des fils d'exécutions est approprié.

Dans certains cas, il peut être judicieux de commencer à traiter une requête de façon synchrone mais de pouvoir déléguer le reste du traitement à un processus enfant si besoin. Ce comportement peut être implémenté en utilisant un serveur synchrone et en laissant à la méthode `handle()`, de la classe gérant les requêtes, le soin de créer le processus enfant explicitement.

Another approach to handling multiple simultaneous requests in an environment that supports neither threads nor `fork()` (or where these are too expensive or inappropriate for the service) is to maintain an explicit table of partially finished requests and to use `selectors` to decide which request to work on next (or whether to handle a new incoming request). This is particularly important for stream services where each client can potentially be connected for a long time (if threads or subprocesses cannot be used).

22.16.2 Objets serveur

class socketserver.**BaseServer**(*server_address*, *RequestHandlerClass*)

Il s'agit de la classe parente de tous les objets serveur du module. Elle déclare l'interface, définie ci-dessous, mais laisse aux classes filles le soin d'implémenter la plupart des méthodes. Les deux paramètres sont stockés respectivement dans les attributs `server_address` et `RequestHandlerClass`.

fileno()

Renvoie un entier représentant le descripteur de fichier pour le connecteur que le serveur écoute. Cette fonction est, la plupart du temps, passée à `selectors` afin de pouvoir surveiller plusieurs serveurs dans le même processus.

handle_request()

Traite une seule requête. Cette fonction appelle, dans l'ordre, les méthodes `get_request()`, `verify_request()` et `process_request()`. Si la méthode `handle()` de la classe de gestion des requêtes lève une exception, alors la méthode `handle_error()` du serveur est appelée. Si aucune requête n'est reçue avant « `timeout` » secondes, `handle_timeout()` est appelée et `handle_request()` rend la main.

serve_forever(*poll_interval=0.5*)

Gère les requêtes indéfiniment jusqu'à ce que `shutdown()` soit appelée. Vérifie si une demande d'arrêt (*shutdown*) a été émise toutes les *poll_interval* secondes. Ignore l'attribut `timeout`. Appelle également `service_actions()`, qui peut être utilisée par une classe enfant ou une classe de mélange afin d'implémenter une action spécifique pour un service donné. Par exemple, la classe `ForkingMixIn` utilise `service_actions()` pour supprimer les processus enfants zombies.

Modifié dans la version 3.3 : La méthode `serve_forever` appelle dorénavant `service_actions`.

service_actions()

Cette méthode est appelée dans la boucle de `serve_forever()`. Cette méthode peut être surchargée par une classe fille ou une classe de mélange afin d'effectuer une action spécifique à un service donné, comme une action de nettoyage.

Ajouté dans la version 3.3.

shutdown()

Demande l'arrêt de la boucle de `serve_forever()` et attend jusqu'à ce que ce soit fait. `shutdown()` doit être appelée dans un fil d'exécution différent de `serve_forever()` sous peine d'interblocage.

server_close()

Nettoie le serveur. Peut être surchargée.

address_family

The family of protocols to which the server's socket belongs. Common examples are `socket.AF_INET`,

`socket.AF_INET6`, and `socket.AF_UNIX`. Subclass the TCP or UDP server classes in this module with class attribute `address_family = AF_INET6` set if you want IPv6 server classes.

RequestHandlerClass

La classe de gestion des requêtes, fournie par l'utilisateur. Une instance de cette classe est créée pour chaque requête.

server_address

Renvoie l'adresse sur laquelle le serveur écoute. Le format de l'adresse dépend de la famille de protocoles utilisée ; pour plus de détails, voir la documentation du module `socket`. Pour les protocoles Internet, cette valeur est une paire formée d'une chaine de caractère pour l'adresse et d'un entier pour le port. Exemple : `('127.0.0.1', 80)`.

socket

L'objet connecteur utilisé par le serveur pour écouter les nouvelles requêtes.

Les classes serveurs prennent en charge les variables de classe suivantes :

allow_reuse_address

Indique si le serveur autorise la réutilisation d'une adresse. La valeur par défaut est `False` mais cela peut être changé dans les classes enfants.

request_queue_size

La taille de la file des requêtes. Lorsque traiter une requête prend du temps, toute nouvelle requête arrivant pendant que le serveur est occupé est placé dans une file jusqu'à atteindre « `request_queue_size` » requêtes. Si la queue est pleine, les nouvelles requêtes clientes se voient renvoyer une erreur *Connection denied*. La valeur par défaut est habituellement 5 mais peut être changée par les classes filles.

socket_type

Le type de connecteur utilisé par le serveur ; `socket.SOCK_STREAM` et `socket.SOCK_DGRAM` sont deux valeurs usuelles.

timeout

Délai d'attente en secondes, ou `None` si aucune limite n'est demandée. Si `handle_request()` ne reçoit aucune requête entrante pendant le délai d'attente, la méthode `handle_timeout()` est appelée.

Il existe plusieurs méthodes serveur pouvant être surchargées par des classes dérivant de classes de base comme `TCPServer` ; ces méthodes ne sont pas utiles aux utilisateurs externes de l'objet serveur.

finish_request(*request*, *client_address*)

Méthode en charge de traiter la requête en instanciant `RequestHandlerClass` et en appelant sa méthode `handle()`.

get_request()

Accepte obligatoirement une requête depuis le connecteur et renvoie une paire contenant le *nouvel* objet connecteur utilisé pour communiquer avec le client et l'adresse du client.

handle_error(*request*, *client_address*)

Cette fonction est appelée si la méthode `handle()` de l'objet `RequestHandlerClass` lève une exception. Par défaut, la méthode imprime la trace d'appels sur la sortie d'erreur standard et continue de traiter les requêtes suivantes.

Modifié dans la version 3.6 : N'est maintenant appelée que sur les exceptions dérivant de la classe `Exception`.

handle_timeout()

Cette fonction est appelée lorsque l'attribut `timeout` est réglé à autre chose que `None` et que le délai d'attente expire sans qu'aucune requête ne soit reçue. Par défaut, cette fonction récupère le statut de tous les processus enfants ayant terminé pour les serveurs utilisant des processus ou ne fait rien pour le cas des serveurs utilisant des fils d'exécution.

process_request(*request*, *client_address*)

Appelle `finish_request()` pour instancier `RequestHandlerClass`. Si désiré, cette fonction peut

créer des processus fils ou des fils d'exécution pour traiter les requêtes ; les classes de mélange `ForkingMixIn` et `ThreadingMixIn` implémentent cela.

server_activate()

Appelée par le constructeur du serveur afin de l'activer. Le comportement par défaut pour un serveur TCP est de seulement invoquer `listen()` sur le connecteur du serveur. Peut être surchargée.

server_bind()

Appelée par le constructeur du serveur afin d'assigner (*bind*) l'adresse requise au connecteur du serveur. Peut être surchargée.

verify_request(*request*, *client_address*)

Doit renvoyer un booléen. Si la valeur est `True`, la requête sera traitée. Si la valeur est `False`, la requête sera refusée. Cette fonction peut être surchargée afin d'implémenter une stratégie de contrôle d'accès au serveur. L'implémentation par défaut renvoie toujours `True`.

Modifié dans la version 3.6 : La gestion du protocole *context manager* a été ajoutée. Sortir du gestionnaire de contexte revient à appeler `server_close()`.

22.16.3 Objets gestionnaire de requêtes

class socketserver.**BaseRequestHandler**

Classe de base de tous les objets gestionnaire de requêtes. Elle déclare l'interface, définie ci-dessous, pour tous les gestionnaires. Une implémentation concrète doit définir une nouvelle méthode `handle()` et peut surcharger n'importe quelle autre méthode. Cette classe concrète est instanciée pour chaque requête.

setup()

Appelée avant la méthode `handle()` afin d'effectuer toutes les opérations d'initialisation requises. L'implémentation par défaut ne fait rien.

handle()

This function must do all the work required to service a request. The default implementation does nothing. Several instance attributes are available to it ; the request is available as `request` ; the client address as `client_address` ; and the server instance as `server`, in case it needs access to per-server information.

The type of `request` is different for datagram or stream services. For stream services, `request` is a socket object ; for datagram services, `request` is a pair of string and socket.

finish()

Appelée après la méthode `handle()` pour effectuer les opérations de nettoyage requises. L'implémentation par défaut ne fait rien. Si `setup()` lève une exception, cette méthode n'est pas appelée.

request

The *new* `socket.socket` object to be used to communicate with the client.

client_address

Client address returned by `BaseServer.get_request()`.

server

`BaseServer` object used for handling the request.

class socketserver.**StreamRequestHandler**

class socketserver.**DatagramRequestHandler**

These `BaseRequestHandler` subclasses override the `setup()` and `finish()` methods, and provide `rfile` and `wfile` attributes.

rfile

A file object from which receives the request is read. Support the `io.BufferedIOBase` readable interface.

wfile

A file object to which the reply is written. Support the `io.BufferedIOBase` writable interface

Modifié dans la version 3.6 : *wfile* also supports the *io.BufferedIOBase* writable interface.

22.16.4 Exemples

Exemple pour socketserver.TCPServer

Implémentation côté serveur :

```
import socketserver

class MyTCPHandler(socketserver.BaseRequestHandler):
    """
    The request handler class for our server.

    It is instantiated once per connection to the server, and must
    override the handle() method to implement communication to the
    client.
    """

    def handle(self):
        # self.request is the TCP socket connected to the client
        pieces = [b'']
        total = 0
        while b'\n' not in pieces[-1] and total < 10_000:
            pieces.append(self.request.recv(2000))
            total += len(pieces[-1])
        self.data = b''.join(pieces)
        print(f"Received from {self.client_address[0]}:")
        print(self.data.decode("utf-8"))
        # just send back the same data, but upper-cased
        self.request.sendall(self.data.upper())
        # after we return, the socket will be closed.

if __name__ == "__main__":
    HOST, PORT = "localhost", 9999

    # Create the server, binding to localhost on port 9999
    with socketserver.TCPServer((HOST, PORT), MyTCPHandler) as server:
        # Activate the server; this will keep running until you
        # interrupt the program with Ctrl-C
        server.serve_forever()
```

Une implémentation alternative du gestionnaire de requêtes utilisant les flux de données (avec des objets fichier-compatibles simplifiant la communication en fournissant l'interface fichier standard) :

```
class MyTCPHandler(socketserver.StreamRequestHandler):

    def handle(self):
        # self.rfile is a file-like object created by the handler.
        # We can now use e.g. readline() instead of raw recv() calls.
        # We limit ourselves to 10000 bytes to avoid abuse by the sender.
        self.data = self.rfile.readline(10000).rstrip()
        print(f"{self.client_address[0]} wrote:")
        print(self.data.decode("utf-8"))
        # Likewise, self.wfile is a file-like object used to write back
        # to the client
        self.wfile.write(self.data.upper())
```

The difference is that the `readline()` call in the second handler will call `recv()` multiple times until it encounters

a newline character, while the the first handler had to use a `recv()` loop to accumulate data until a newline itself. If it had just used a single `recv()` without the loop it would just have returned what has been received so far from the client. TCP is stream based : data arrives in the order it was sent, but there no correlation between client `send()` or `sendall()` calls and the number of `recv()` calls on the server required to receive it.

Implémentation côté client :

```
import socket
import sys

HOST, PORT = "localhost", 9999
data = " ".join(sys.argv[1:])

# Create a socket (SOCK_STREAM means a TCP socket)
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
    # Connect to server and send data
    sock.connect((HOST, PORT))
    sock.sendall(bytes(data, "utf-8"))
    sock.sendall(b"\n")

    # Receive data from the server and shut down
    received = str(sock.recv(1024), "utf-8")

print("Sent:     ", data)
print("Received:", received)
```

La sortie de cet exemple devrait ressembler à ça :

Serveur :

```
$ python TCPServer.py
127.0.0.1 wrote:
b'hello world with TCP'
127.0.0.1 wrote:
b'python is nice'
```

Client :

```
$ python TCPClient.py hello world with TCP
Sent:     hello world with TCP
Received: HELLO WORLD WITH TCP
$ python TCPClient.py python is nice
Sent:     python is nice
Received: PYTHON IS NICE
```

Exemple pour `socketserver.UDPServer`

Implémentation côté serveur :

```
import socketserver

class MyUDPHandler(socketserver.BaseRequestHandler):
    """
    This class works similar to the TCP handler class, except that
    self.request consists of a pair of data and client socket, and since
    there is no connection the client address must be given explicitly
    when sending data back via sendto().
    """
```

(suite sur la page suivante)

(suite de la page précédente)

```
    def handle(self):
        data = self.request[0].strip()
        socket = self.request[1]
        print(f"{self.client_address[0]} wrote:")
        print(data)
        socket.sendto(data.upper(), self.client_address)

if __name__ == "__main__":
    HOST, PORT = "localhost", 9999
    with socketserver.UDPServer((HOST, PORT), MyUDPHandler) as server:
        server.serve_forever()
```

Implémentation côté client :

```
import socket
import sys

HOST, PORT = "localhost", 9999
data = " ".join(sys.argv[1:])

# SOCK_DGRAM is the socket type to use for UDP sockets
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# As you can see, there is no connect() call; UDP has no connections.
# Instead, data is directly sent to the recipient via sendto().
sock.sendto(bytes(data + "\n", "utf-8"), (HOST, PORT))
received = str(sock.recv(1024), "utf-8")

print("Sent:     ", data)
print("Received:", received)
```

La sortie de cet exemple devrait ressembler exactement à la sortie de l'exemple pour le serveur TCP.

Classes de mélange asynchrone

Pour développer des gestionnaires asynchrones, utilisez les classes `ThreadingMixIn` et `ForkingMixIn`.

Exemple pour `ThreadingMixIn` :

```
import socket
import threading
import socketserver

class ThreadedTCPRequestHandler(socketserver.BaseRequestHandler):

    def handle(self):
        data = str(self.request.recv(1024), 'ascii')
        cur_thread = threading.current_thread()
        response = bytes("{}: {}".format(cur_thread.name, data), 'ascii')
        self.request.sendall(response)

class ThreadedTCPServer(socketserver.ThreadingMixIn, socketserver.TCPServer):
    pass

def client(ip, port, message):
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
        sock.connect((ip, port))
```

(suite sur la page suivante)

(suite de la page précédente)

```
        sock.sendall(bytes(message, 'ascii'))
        response = str(sock.recv(1024), 'ascii')
        print("Received: {}".format(response))

if __name__ == "__main__":
    # Port 0 means to select an arbitrary unused port
    HOST, PORT = "localhost", 0

    server = ThreadedTCPServer((HOST, PORT), ThreadedTCPRequestHandler)
    with server:
        ip, port = server.server_address

        # Start a thread with the server -- that thread will then start one
        # more thread for each request
        server_thread = threading.Thread(target=server.serve_forever)
        # Exit the server thread when the main thread terminates
        server_thread.daemon = True
        server_thread.start()
        print("Server loop running in thread:", server_thread.name)

        client(ip, port, "Hello World 1")
        client(ip, port, "Hello World 2")
        client(ip, port, "Hello World 3")

        server.shutdown()
```

La sortie de cet exemple devrait ressembler à ça :

```
$ python ThreadedTCPServer.py
Server loop running in thread: Thread-1
Received: Thread-2: Hello World 1
Received: Thread-3: Hello World 2
Received: Thread-4: Hello World 3
```

La classe `ForkingMixIn` est utilisable de la même façon à la différence près que le serveur crée un nouveau processus fils pour chaque requête. Disponible uniquement sur les plateformes POSIX prenant en charge `fork()`.

22.17 `http.server` --- HTTP servers

Code source : Lib/http/server.py

Ce module définit des classes implémentant des serveurs HTTP.

> **Avertissement**
>
> `http.server` is not recommended for production. It only implements *basic security checks*.

Availability : not WASI.

This module does not work or is not available on WebAssembly. See *Plateformes WebAssembly* for more information.

Une des classes, `HTTPServer`, est une sous-classe de `socketserver.TCPServer`. Elle crée une interface de connexion (*socket* en anglais) avant de rester à l'écoute des messages reçus sur celle-ci, les répartissant à un gestionnaire d'évènements (*handler* en anglais). Le code pour créer et exécuter le serveur ressemble à ceci :

```
def run(server_class=HTTPServer, handler_class=BaseHTTPRequestHandler):
    server_address = ('', 8000)
    httpd = server_class(server_address, handler_class)
    httpd.serve_forever()
```

class `http.server.`**`HTTPServer`**(*server_address*, *RequestHandlerClass*)

Cette classe hérite de la classe `TCPServer`. Ses instances contiennent l'adresse du serveur dans les variables d'instance `server_name` et `server_port`.Le serveur est accessible par le gestionnaire d'évènements, habituellement par le biais de sa variable d'instance `server`.

class `http.server.`**`ThreadingHTTPServer`**(*server_address*, *RequestHandlerClass*)

Cette classe est identique à *HTTPServer* mais utilise des fils d'exécution afin de gérer les requêtes, et ce, par le biais de `ThreadingMixIn`. Ceci est utile afin de gérer les pré-ouvertures des interfaces de connexion des navigateurs web, sur lesquelles `HTTPServer` attendrait de façon perpétuelle.

Ajouté dans la version 3.7.

On doit passer un *RequestHandlerClass* lors de l'instanciation de `HTTPServer` et de `ThreadingHTTPServer`. Ce module fournit trois variantes différentes :

class `http.server.`**`BaseHTTPRequestHandler`**(*request*, *client_address*, *server*)

Cette classe est utilisée afin de gérer les requêtes HTTP arrivant au serveur. Elle ne peut pas répondre par elle-même à des requêtes HTTP ; cette méthode doit être surchargée dans les classes dérivées, par exemple *GET* ou *POST*. La classe `BaseHTTPRequestHandler` fournit plusieurs classes et variables d'instance, ainsi que des méthodes à utiliser par les sous-classes.

The handler will parse the request and the headers, then call a method specific to the request type. The method name is constructed from the request. For example, for the request method `SPAM`, the `do_SPAM()` method will be called with no arguments. All of the relevant information is stored in instance variables of the handler. Subclasses should not need to override or extend the `__init__()` method.

`BaseHTTPRequestHandler` a les variables d'instances suivantes :

`client_address`

Contient un *n*-uplet de la forme `(host, port)`, faisant référence à l'adresse du client.

`server`

Contient l'instance du serveur.

`close_connection`

Booléen qui doit être défini avant que `handle_one_request()` ne termine son exécution, indiquant si on peut recevoir une autre requête ou si la connexion doit être fermée.

`requestline`

Contient la chaîne de caractères représentant la ligne de requête HTTP. La dénotation de fin de ligne *CRLF* est enlevée. Cet attribut doit être défini par `handle_one_request()`. Dans le cas où aucune ligne de requête valide n'a été traitée, il doit prendre la valeur de la chaîne de caractères vide.

`command`

Contient la commande (le type de requête). Par exemple, `'GET'`.

`path`

Contient le chemin de la requête. Si la composante de requête de l'URL est présente, alors `path` contient la requête. Selon la terminologie de **RFC 3986**, `path` inclut ici `hier-part` et la `query`.

`request_version`

Contient la version de la requête, en chaîne de caractères. Par exemple, `'HTTP/1.0'`.

`headers`

Holds an instance of the class specified by the `MessageClass` class variable. This instance parses and manages the headers in the HTTP request. The `parse_headers()` function from `http.client` is used to parse the headers and it requires that the HTTP request provide a valid **RFC 2822** style header.

rfile

An *io.BufferedIOBase* input stream, ready to read from the start of the optional input data.

wfile

Contains the output stream for writing a response back to the client. Proper adherence to the HTTP protocol must be used when writing to this stream in order to achieve successful interoperation with HTTP clients.

Modifié dans la version 3.6 : This is an *io.BufferedIOBase* stream.

BaseHTTPRequestHandler has the following attributes :

server_version

Précise la version du logiciel du serveur. Vous pouvez la modifier. Le format utilisé est constitué de plusieurs chaînes de caractères séparées par des caractères d'espacement, où chaque chaîne prend la forme *nom[/version]*. Par exemple, `'BaseHTTP/0.2'`.

sys_version

Contient la version système de Python, dans une forme pouvant être utilisée par la méthode *version_string* ainsi que par la variable de classe *server_version*. Par exemple, `'Python/1.4'`.

error_message_format

Définit une chaîne de caractères avec champs de formatage devant être utilisée par la méthode *send_error()* afin de construire une réponse d'erreur pour le client. Par défaut, la chaîne contient des variables provenant de l'attribut *responses* se basant sur le code de statut passé à *send_error()*.

error_content_type

Définit l'en-tête HTTP Content-Type des réponses d'erreur envoyées au client. La valeur par défaut est `'text/html'`.

protocol_version

Specifies the HTTP version to which the server is conformant. It is sent in responses to let the client know the server's communication capabilities for future requests. If set to `'HTTP/1.1'`, the server will permit HTTP persistent connections ; however, your server *must* then include an accurate `Content-Length` header (using *send_header()*) in all of its responses to clients. For backwards compatibility, the setting defaults to `'HTTP/1.0'`.

MessageClass

Définit une classe similaire à *email.message.Message* permettant l'analyse des en-têtes HTTP. Habituellement, cette valeur n'est pas modifiée, et prend par défaut la valeur de *http.client.HTTPMessage*.

responses

Cet attribut contient une table de correspondance entre des codes d'erreurs dénotés par des entiers et des *n*-uplets contenant un message court et un message long. Par exemple, `{code: (shortmessage, longmessage)}`. Habituellement, le message *shortmessage* correspond à la clé *message* d'une réponse d'erreur, alors que le message *longmessage* correspond à la clé *explain* de celle-ci. Il est utilisé par les méthodes *send_response_only()* et *send_error()*.

Une instance de la classe *BaseHTTPRequestHandler* contient les méthodes suivantes :

handle()

Calls *handle_one_request()* once (or, if persistent connections are enabled, multiple times) to handle incoming HTTP requests. You should never need to override it ; instead, implement appropriate `do_*()` methods.

handle_one_request()

This method will parse and dispatch the request to the appropriate `do_*()` method. You should never need to override it.

handle_expect_100()

When an HTTP/1.1 conformant server receives an `Expect: 100-continue` request header it responds back with a `100 Continue` followed by `200 OK` headers. This method can be overridden to

raise an error if the server does not want the client to continue. For e.g. server can choose to send `417 Expectation Failed` as a response header and `return False`.

Ajouté dans la version 3.2.

send_error(*code*, *message=None*, *explain=None*)

Sends and logs a complete error reply to the client. The numeric *code* specifies the HTTP error code, with *message* as an optional, short, human readable description of the error. The *explain* argument can be used to provide more detailed information about the error ; it will be formatted using the `error_message_format` attribute and emitted, after a complete set of headers, as the response body. The `responses` attribute holds the default values for *message* and *explain* that will be used if no value is provided ; for unknown codes the default value for both is the string `???`. The body will be empty if the method is HEAD or the response code is one of the following : `1xx`, `204 No Content`, `205 Reset Content`, `304 Not Modified`.

Modifié dans la version 3.4 : The error response includes a Content-Length header. Added the *explain* argument.

send_response(*code*, *message=None*)

Adds a response header to the headers buffer and logs the accepted request. The HTTP response line is written to the internal buffer, followed by *Server* and *Date* headers. The values for these two headers are picked up from the `version_string()` and `date_time_string()` methods, respectively. If the server does not intend to send any other headers using the `send_header()` method, then `send_response()` should be followed by an `end_headers()` call.

Modifié dans la version 3.3 : Headers are stored to an internal buffer and `end_headers()` needs to be called explicitly.

send_header(*keyword*, *value*)

Adds the HTTP header to an internal buffer which will be written to the output stream when either `end_headers()` or `flush_headers()` is invoked. *keyword* should specify the header keyword, with *value* specifying its value. Note that, after the send_header calls are done, `end_headers()` MUST BE called in order to complete the operation.

Modifié dans la version 3.2 : Headers are stored in an internal buffer.

send_response_only(*code*, *message=None*)

Sends the response header only, used for the purposes when `100 Continue` response is sent by the server to the client. The headers not buffered and sent directly the output stream.If the *message* is not specified, the HTTP message corresponding the response *code* is sent.

Ajouté dans la version 3.2.

end_headers()

Adds a blank line (indicating the end of the HTTP headers in the response) to the headers buffer and calls `flush_headers()`.

Modifié dans la version 3.2 : The buffered headers are written to the output stream.

flush_headers()

Finally send the headers to the output stream and flush the internal headers buffer.

Ajouté dans la version 3.3.

log_request(*code='-'*, *size='-'*)

Logs an accepted (successful) request. *code* should specify the numeric HTTP code associated with the response. If a size of the response is available, then it should be passed as the *size* parameter.

log_error(...)

Logs an error when a request cannot be fulfilled. By default, it passes the message to `log_message()`, so it takes the same arguments (*format* and additional values).

log_message(*format*, ...)

Logs an arbitrary message to `sys.stderr`. This is typically overridden to create custom error logging

mechanisms. The *format* argument is a standard printf-style format string, where the additional arguments to `log_message()` are applied as inputs to the formatting. The client ip address and current date and time are prefixed to every message logged.

version_string()

Returns the server software's version string. This is a combination of the `server_version` and `sys_version` attributes.

date_time_string(*timestamp=None*)

Returns the date and time given by *timestamp* (which must be `None` or in the format returned by `time.time()`), formatted for a message header. If *timestamp* is omitted, it uses the current date and time.

The result looks like `'Sun, 06 Nov 1994 08:49:37 GMT'`.

log_date_time_string()

Returns the current date and time, formatted for logging.

address_string()

Returns the client address.

Modifié dans la version 3.3 : Previously, a name lookup was performed. To avoid name resolution delays, it now always returns the IP address.

class `http.server.`**SimpleHTTPRequestHandler**(*request*, *client_address*, *server*, *directory=None*)

This class serves files from the directory *directory* and below, or the current directory if *directory* is not provided, directly mapping the directory structure to HTTP requests.

Modifié dans la version 3.7 : Added the *directory* parameter.

Modifié dans la version 3.9 : The *directory* parameter accepts a *path-like object*.

A lot of the work, such as parsing the request, is done by the base class `BaseHTTPRequestHandler`. This class implements the `do_GET()` and `do_HEAD()` functions.

The following are defined as class-level attributes of `SimpleHTTPRequestHandler` :

server_version

This will be `"SimpleHTTP/" + __version__`, where `__version__` is defined at the module level.

extensions_map

A dictionary mapping suffixes into MIME types, contains custom overrides for the default system mappings. The mapping is used case-insensitively, and so should contain only lower-cased keys.

Modifié dans la version 3.9 : This dictionary is no longer filled with the default system mappings, but only contains overrides.

The `SimpleHTTPRequestHandler` class defines the following methods :

do_HEAD()

This method serves the `'HEAD'` request type : it sends the headers it would send for the equivalent `GET` request. See the `do_GET()` method for a more complete explanation of the possible headers.

do_GET()

The request is mapped to a local file by interpreting the request as a path relative to the current working directory.

If the request was mapped to a directory, the directory is checked for a file named `index.html` or `index.htm` (in that order). If found, the file's contents are returned ; otherwise a directory listing is generated by calling the `list_directory()` method. This method uses `os.listdir()` to scan the directory, and returns a `404` error response if the `listdir()` fails.

If the request was mapped to a file, it is opened. Any `OSError` exception in opening the requested file is mapped to a `404`, `'File not found'` error. If there was an `'If-Modified-Since'` header in the request, and the file was not modified after this time, a `304`, `'Not Modified'` response is sent. Otherwise, the content type is guessed by calling the `guess_type()` method, which in turn uses the *extensions_map* variable, and the file contents are returned.

A `'Content-type:'` header with the guessed content type is output, followed by a `'Content-Length:'` header with the file's size and a `'Last-Modified:'` header with the file's modification time.

Then follows a blank line signifying the end of the headers, and then the contents of the file are output. If the file's MIME type starts with `text/` the file is opened in text mode ; otherwise binary mode is used.

For example usage, see the implementation of the `test` function in Lib/http/server.py.

Modifié dans la version 3.7 : Support of the `'If-Modified-Since'` header.

The *`SimpleHTTPRequestHandler`* class can be used in the following manner in order to create a very basic web-server serving files relative to the current directory :

```
import http.server
import socketserver

PORT = 8000

Handler = http.server.SimpleHTTPRequestHandler

with socketserver.TCPServer(("", PORT), Handler) as httpd:
    print("serving at port", PORT)
    httpd.serve_forever()
```

`SimpleHTTPRequestHandler` can also be subclassed to enhance behavior, such as using different index file names by overriding the class attribute `index_pages`.

`http.server` can also be invoked directly using the `-m` switch of the interpreter. Similar to the previous example, this serves files relative to the current directory :

```
python -m http.server
```

The server listens to port 8000 by default. The default can be overridden by passing the desired port number as an argument :

```
python -m http.server 9000
```

By default, the server binds itself to all interfaces. The option `-b/--bind` specifies a specific address to which it should bind. Both IPv4 and IPv6 addresses are supported. For example, the following command causes the server to bind to localhost only :

```
python -m http.server --bind 127.0.0.1
```

Modifié dans la version 3.4 : Added the `--bind` option.

Modifié dans la version 3.8 : Support IPv6 in the `--bind` option.

By default, the server uses the current directory. The option `-d/--directory` specifies a directory to which it should serve the files. For example, the following command uses a specific directory :

```
python -m http.server --directory /tmp/
```

Modifié dans la version 3.7 : Added the `--directory` option.

By default, the server is conformant to HTTP/1.0. The option `-p/--protocol` specifies the HTTP version to which the server is conformant. For example, the following command runs an HTTP/1.1 conformant server :

```
python -m http.server --protocol HTTP/1.1
```

Modifié dans la version 3.11 : Added the `--protocol` option.

class `http.server.`**`CGIHTTPRequestHandler`**(*request*, *client_address*, *server*)

This class is used to serve either files or output of CGI scripts from the current directory and below. Note that mapping HTTP hierarchic structure to local directory structure is exactly as in `SimpleHTTPRequestHandler`.

> **Note**
>
> CGI scripts run by the `CGIHTTPRequestHandler` class cannot execute redirects (HTTP code 302), because code 200 (script output follows) is sent prior to execution of the CGI script. This pre-empts the status code.

The class will however, run the CGI script, instead of serving it as a file, if it guesses it to be a CGI script. Only directory-based CGI are used --- the other common server configuration is to treat special extensions as denoting CGI scripts.

The `do_GET()` and `do_HEAD()` functions are modified to run CGI scripts and serve the output, instead of serving files, if the request leads to somewhere below the `cgi_directories` path.

The `CGIHTTPRequestHandler` defines the following data member :

`cgi_directories`

This defaults to `['/cgi-bin', '/htbin']` and describes directories to treat as containing CGI scripts.

The `CGIHTTPRequestHandler` defines the following method :

`do_POST`()

This method serves the `'POST'` request type, only allowed for CGI scripts. Error 501, "Can only POST to CGI scripts", is output when trying to POST to a non-CGI url.

Note that CGI scripts will be run with UID of user nobody, for security reasons. Problems with the CGI script will be translated to error 403.

Deprecated since version 3.13, will be removed in version 3.15 : `CGIHTTPRequestHandler` is being removed in 3.15. CGI has not been considered a good way to do things for well over a decade. This code has been unmaintained for a while now and sees very little practical use. Retaining it could lead to further *security considerations*.

`CGIHTTPRequestHandler` can be enabled in the command line by passing the `--cgi` option :

```
python -m http.server --cgi
```

Deprecated since version 3.13, will be removed in version 3.15 : `http.server` command line `--cgi` support is being removed because `CGIHTTPRequestHandler` is being removed.

> **Avertissement**
>
> `CGIHTTPRequestHandler` and the `--cgi` command line option are not intended for use by untrusted clients and may be vulnerable to exploitation. Always use within a secure environment.

22.17.1 Considérations de sécurité

`SimpleHTTPRequestHandler` will follow symbolic links when handling requests, this makes it possible for files outside of the specified directory to be served.

Earlier versions of Python did not scrub control characters from the log messages emitted to stderr from `python -m http.server` or the default `BaseHTTPRequestHandler.log_message` implementation. This could allow remote clients connecting to your server to send nefarious control codes to your terminal.

Modifié dans la version 3.12 : Control characters are scrubbed in stderr logs.

22.18 `http.cookies` --- HTTP state management

Code source : Lib/http/cookies.py

Le module `http.cookies` définit des classes abstrayant le concept de témoin web (cookie), un mécanisme de gestion d'état pour HTTP. Il fournit une abstraction gérant des données textuelles et tout type de données sérialisable comme valeur de témoin.

The module formerly strictly applied the parsing rules described in the **RFC 2109** and **RFC 2068** specifications. It has since been discovered that MSIE 3.0x didn't follow the character rules outlined in those specs ; many current-day browsers and servers have also relaxed parsing rules when it comes to cookie handling. As a result, this module now uses parsing rules that are a bit less strict than they once were.

The character set, `string.ascii_letters`, `string.digits` and `!#$%&'*+-.^_` `|~:` denote the set of valid characters allowed by this module in a cookie name (as `key`).

Modifié dans la version 3.3 : Allowed ' :' as a valid cookie name character.

> **Note**
>
> Quand un témoin invalide est rencontré, l'exception `CookieError` est levée. Si les données du témoin proviennent d'un navigateur il faut impérativement gérer les données invalides en attrapant `CookieError`.

exception `http.cookies.`**`CookieError`**

Exception levée pour cause d'incompatibilité avec la **RFC 2109**. Exemples : attributs incorrects, en-tête `Set-Cookie` incorrect, etc.

class `http.cookies.`**`BaseCookie`**(*[input]*)

Cette classe définit un dictionnaire dont les clés sont des chaines de caractères et dont les valeurs sont des instances de `Morsel`. Notez qu'à l'assignation d'une valeur à une clé, la valeur est transformée en `Morsel` contenant la clé et la valeur.

Si l'argument *input* est donné, il est passé à la méthode `load()`.

class `http.cookies.`**`SimpleCookie`**(*[input]*)

This class derives from `BaseCookie` and overrides `value_decode()` and `value_encode()`. `SimpleCookie` supports strings as cookie values. When setting the value, `SimpleCookie` calls the builtin `str()` to convert the value to a string. Values received from HTTP are kept as strings.

> **Voir aussi**
>
> **Module `http.cookiejar`**
> Gestion de témoins HTTP pour *clients* web. Les modules `http.cookiejar` et `http.cookies` ne dépendent pas l'un de l'autre.
>
> **RFC 2109 - HTTP State Management Mechanism**
> Spécification de gestion d'états implantée par ce module.

22.18.1 Objets *Cookie*

`BaseCookie.`**`value_decode`**(*val*)

Renvoie une paire `(real_value, coded_value)` depuis une représentation de chaine. `real_value` peut être de n'importe quel type. Cette méthode ne décode rien dans `BaseCookie` – elle existe pour être surchargée.

`BaseCookie.`**`value_encode`**(*val*)

Renvoie une paire `(real_value, coded_value)`. *val* peut être de n'importe quel type, mais

`coded_value` est toujours converti en chaine de caractères. Cette méthode n'encode pas dans `BaseCookie` – elle existe pour être surchargée.

Généralement, les méthodes `value_encode()` et `value_decode()` doivent être inverses l'une de l'autre, c'est-à-dire qu'en envoyant la sortie de l'un dans l'entrée de l'autre la valeur finale doit être égale à la valeur initiale.

`BaseCookie.output`(*attrs=None*, *header='Set-Cookie :'*, *sep='\r\n'*)

Return a string representation suitable to be sent as HTTP headers. *attrs* and *header* are sent to each `Morsel`'s `output()` method. *sep* is used to join the headers together, and is by default the combination `'\r\n'` (CRLF).

`BaseCookie.js_output`(*attrs=None*)

Renvoie un extrait de code JavaScript qui, lorsque exécuté par un navigateur qui supporte le JavaScript, va fonctionner de la même manière que si les en-têtes HTTP avaient été envoyés.

attrs a la même signification que dans la méthode `output()`.

`BaseCookie.load`(*rawdata*)

Si *rawdata* est une chaine de caractères, l'analyser comme étant un `HTTP_COOKIE` et ajouter les valeurs trouvées en tant que `Morsel`s. S'il s'agit d'un dictionnaire, cela est équivalent à :

```
for k, v in rawdata.items():
    cookie[k] = v
```

22.18.2 Objets *Morsel*

class `http.cookies.Morsel`

Abstraction de paire clé / valeur, accompagnée d'attributs provenant de la spécification **RFC 2109**.

Morsels are dictionary-like objects, whose set of keys is constant --- the valid **RFC 2109** attributes, which are :

expires
path
comment
domain
max-age
secure
version
httponly
samesite
partitioned

L'attribut `httponly` spécifie que le témoin transféré dans les requêtes HTTP n'est pas accessible par le biais de JavaScript. Il s'agit d'une contre-mesure à certaines attaques de scripts inter-sites (*XSS*).

L'attribut `samesite` spécifie que le navigateur n'est pas autorisé à envoyer le témoin de connexion avec les requêtes inter-sites. Cela vise à contrer les attaques *CSRF*. Les valeurs valides pour cet attribut sont « Strict » et « Lax ».

The attribute `partitioned` indicates to user agents that these cross-site cookies *should* only be available in the same top-level context that the cookie was first set in. For this to be accepted by the user agent, you **must** also set `Secure`.

In addition, it is recommended to use the `__Host` prefix when setting partitioned cookies to make them bound to the hostname and not the registrable domain. Read CHIPS (Cookies Having Independent Partitioned State) for full details and examples.

Les clés ne sont pas sensibles à la casse, leur valeur par défaut est `''`.

Modifié dans la version 3.5 : `__eq__()` now takes `key` and `value` into account.

Modifié dans la version 3.7 : les attributs `key`, `value` et `coded_value` sont en lecture seule. Utilisez `set()` pour les assigner.

Modifié dans la version 3.8 : ajout de la prise en charge de l'attribut `samesite`.

Modifié dans la version 3.14 : Added support for the `partitioned` attribute.

`Morsel.`**`value`**

La valeur du témoin.

`Morsel.`**`coded_value`**

La valeur codée du témoin. C'est celle qui doit être transférée.

`Morsel.`**`key`**

Le nom du témoin.

`Morsel.`**`set`**(*key*, *value*, *coded_value*)

Assigne les attributs *key*, *value* et *coded_value*.

`Morsel.`**`isReservedKey`**(*K*)

Renvoie si *K* est membre des clés d'un `Morsel`.

`Morsel.`**`output`**(*attrs=None*, *header='Set-Cookie :'*)

Renvoie une représentation textuelle du *Morsel* compatible avec les en-têtes HTTP. Par défaut, tous les attributs sont inclus, à moins que *attrs* ne soit renseigné. Dans ce cas la valeur doit être une liste d'attributs à utiliser. Par défaut, *header* a la valeur `"Set-Cookie:"`.

`Morsel.`**`js_output`**(*attrs=None*)

Renvoie un extrait de code JavaScript qui, lorsque exécuté par un navigateur qui supporte le JavaScript, va fonctionner de la même manière que si les en-têtes HTTP avaient été envoyés.

attrs a la même signification que dans la méthode `output()`.

`Morsel.`**`OutputString`**(*attrs=None*)

Renvoie une chaine de caractères représentant le *Morsel*, nettoyé de son contexte HTTP ou JavaScript.

attrs a la même signification que dans la méthode `output()`.

`Morsel.`**`update`**(*values*)

Met à jour les valeurs du dictionnaire du *Morsel* avec les valeurs provenant du dictionnaire *values*. Lève une erreur si une des clés n'est pas un attribut **RFC 2109** valide.

Modifié dans la version 3.5 : une erreur est levée pour les clés invalides.

`Morsel.`**`copy`**(*value*)

Renvoie une copie superficielle de l'objet *Morsel*.

Modifié dans la version 3.5 : renvoie un objet *Morsel* au lieu d'un `dict`.

`Morsel.`**`setdefault`**(*key*, *value=None*)

Lève une erreur si la clé n'est pas un attribut **RFC 2109** valide, sinon fonctionne de la même manière que `dict.setdefault()`.

22.18.3 Exemple

L'exemple suivant montre comment utiliser le module `http.cookies`.

```
>>> from http import cookies
>>> C = cookies.SimpleCookie()
>>> C["fig"] = "newton"
>>> C["sugar"] = "wafer"
>>> print(C) # generate HTTP headers
Set-Cookie: fig=newton
Set-Cookie: sugar=wafer
>>> print(C.output()) # same thing
Set-Cookie: fig=newton
```

(suite sur la page suivante)

(suite de la page précédente)

```
Set-Cookie: sugar=wafer
>>> C = cookies.SimpleCookie()
>>> C["rocky"] = "road"
>>> C["rocky"]["path"] = "/cookie"
>>> print(C.output(header="Cookie:"))
Cookie: rocky=road; Path=/cookie
>>> print(C.output(attrs=[], header="Cookie:"))
Cookie: rocky=road
>>> C = cookies.SimpleCookie()
>>> C.load("chips=ahoy; vienna=finger") # load from a string (HTTP header)
>>> print(C)
Set-Cookie: chips=ahoy
Set-Cookie: vienna=finger
>>> C = cookies.SimpleCookie()
>>> C.load('keebler="E=everybody; L=\\"Loves\\"; fudge=\\012;";')
>>> print(C)
Set-Cookie: keebler="E=everybody; L=\"Loves\"; fudge=\012;"
>>> C = cookies.SimpleCookie()
>>> C["oreo"] = "doublestuff"
>>> C["oreo"]["path"] = "/"
>>> print(C)
Set-Cookie: oreo=doublestuff; Path=/
>>> C = cookies.SimpleCookie()
>>> C["twix"] = "none for you"
>>> C["twix"].value
'none for you'
>>> C = cookies.SimpleCookie()
>>> C["number"] = 7 # equivalent to C["number"] = str(7)
>>> C["string"] = "seven"
>>> C["number"].value
'7'
>>> C["string"].value
'seven'
>>> print(C)
Set-Cookie: number=7
Set-Cookie: string=seven
```

22.19 `http.cookiejar` --- Cookie handling for HTTP clients

Source code : Lib/http/cookiejar.py

The *`http.cookiejar`* module defines classes for automatic handling of HTTP cookies. It is useful for accessing web sites that require small pieces of data -- *cookies* -- to be set on the client machine by an HTTP response from a web server, and then returned to the server in later HTTP requests.

Both the regular Netscape cookie protocol and the protocol defined by **RFC 2965** are handled. RFC 2965 handling is switched off by default. **RFC 2109** cookies are parsed as Netscape cookies and subsequently treated either as Netscape or RFC 2965 cookies according to the 'policy' in effect. Note that the great majority of cookies on the internet are Netscape cookies. *`http.cookiejar`* attempts to follow the de-facto Netscape cookie protocol (which differs substantially from that set out in the original Netscape specification), including taking note of the `max-age` and `port` cookie-attributes introduced with RFC 2965.

Note

> The various named parameters found in `Set-Cookie` and `Set-Cookie2` headers (eg. `domain` and `expires`) are conventionally referred to as *attributes*. To distinguish them from Python attributes, the documentation for this module uses the term *cookie-attribute* instead.

The module defines the following exception :

exception `http.cookiejar.`**`LoadError`**

Instances of `FileCookieJar` raise this exception on failure to load cookies from a file. `LoadError` is a subclass of `OSError`.

Modifié dans la version 3.3 : `LoadError` used to be a subtype of `IOError`, which is now an alias of `OSError`.

The following classes are provided :

class `http.cookiejar.`**`CookieJar`**(*policy=None*)

policy is an object implementing the `CookiePolicy` interface.

The `CookieJar` class stores HTTP cookies. It extracts cookies from HTTP requests, and returns them in HTTP responses. `CookieJar` instances automatically expire contained cookies when necessary. Subclasses are also responsible for storing and retrieving cookies from a file or database.

class `http.cookiejar.`**`FileCookieJar`**(*filename=None*, *delayload=None*, *policy=None*)

policy is an object implementing the `CookiePolicy` interface. For the other arguments, see the documentation for the corresponding attributes.

A `CookieJar` which can load cookies from, and perhaps save cookies to, a file on disk. Cookies are **NOT** loaded from the named file until either the `load()` or `revert()` method is called. Subclasses of this class are documented in section *FileCookieJar subclasses and co-operation with web browsers*.

This should not be initialized directly – use its subclasses below instead.

Modifié dans la version 3.8 : The filename parameter supports a *path-like object*.

class `http.cookiejar.`**`CookiePolicy`**

This class is responsible for deciding whether each cookie should be accepted from / returned to the server.

class `http.cookiejar.`**`DefaultCookiePolicy`**(*blocked_domains=None*, *allowed_domains=None*, *netscape=True*, *rfc2965=False*, *rfc2109_as_netscape=None*, *hide_cookie2=False*, *strict_domain=False*, *strict_rfc2965_unverifiable=True*, *strict_ns_unverifiable=False*, *strict_ns_domain=DefaultCookiePolicy.DomainLiberal*, *strict_ns_set_initial_dollar=False*, *strict_ns_set_path=False*, *secure_protocols=('https', 'wss')*)

Constructor arguments should be passed as keyword arguments only. *blocked_domains* is a sequence of domain names that we never accept cookies from, nor return cookies to. *allowed_domains* if not `None`, this is a sequence of the only domains for which we accept and return cookies. *secure_protocols* is a sequence of protocols for which secure cookies can be added to. By default *https* and *wss* (secure websocket) are considered secure protocols. For all other arguments, see the documentation for `CookiePolicy` and `DefaultCookiePolicy` objects.

`DefaultCookiePolicy` implements the standard accept / reject rules for Netscape and **RFC 2965** cookies. By default, **RFC 2109** cookies (ie. cookies received in a `Set-Cookie` header with a version cookie-attribute of 1) are treated according to the RFC 2965 rules. However, if RFC 2965 handling is turned off or `rfc2109_as_netscape` is `True`, RFC 2109 cookies are 'downgraded' by the `CookieJar` instance to Netscape cookies, by setting the `version` attribute of the `Cookie` instance to 0. `DefaultCookiePolicy` also provides some parameters to allow some fine-tuning of policy.

class `http.cookiejar.`**`Cookie`**

This class represents Netscape, **RFC 2109** and **RFC 2965** cookies. It is not expected that users of `http.cookiejar` construct their own `Cookie` instances. Instead, if necessary, call `make_cookies()` on a `CookieJar` instance.

Voir aussi

Module `urllib.request`
URL opening with automatic cookie handling.

Module `http.cookies`
HTTP cookie classes, principally useful for server-side code. The `http.cookiejar` and `http.cookies` modules do not depend on each other.

https://curl.se/rfc/cookie_spec.html
The specification of the original Netscape cookie protocol. Though this is still the dominant protocol, the 'Netscape cookie protocol' implemented by all the major browsers (and `http.cookiejar`) only bears a passing resemblance to the one sketched out in `cookie_spec.html`.

RFC 2109 - HTTP State Management Mechanism
Obsoleted by **RFC 2965**. Uses `Set-Cookie` with version=1.

RFC 2965 - HTTP State Management Mechanism
The Netscape protocol with the bugs fixed. Uses `Set-Cookie2` in place of `Set-Cookie`. Not widely used.

https://kristol.org/cookie/errata.html
Unfinished errata to **RFC 2965**.

RFC 2964 - Use of HTTP State Management

22.19.1 CookieJar and FileCookieJar Objects

`CookieJar` objects support the *iterator* protocol for iterating over contained `Cookie` objects.

`CookieJar` has the following methods :

`CookieJar.`**`add_cookie_header`**(*request*)

Add correct `Cookie` header to *request*.

If policy allows (ie. the `rfc2965` and `hide_cookie2` attributes of the `CookieJar`'s `CookiePolicy` instance are true and false respectively), the `Cookie2` header is also added when appropriate.

The *request* object (usually a `urllib.request.Request` instance) must support the methods `get_full_url()`, `has_header()`, `get_header()`, `header_items()`, `add_unredirected_header()` and the attributes `host`, `type`, `unverifiable` and `origin_req_host` as documented by `urllib.request`.

Modifié dans la version 3.3 : *request* object needs `origin_req_host` attribute. Dependency on a deprecated method `get_origin_req_host()` has been removed.

`CookieJar.`**`extract_cookies`**(*response*, *request*)

Extract cookies from HTTP *response* and store them in the `CookieJar`, where allowed by policy.

The `CookieJar` will look for allowable `Set-Cookie` and `Set-Cookie2` headers in the *response* argument, and store cookies as appropriate (subject to the `CookiePolicy.set_ok()` method's approval).

The *response* object (usually the result of a call to `urllib.request.urlopen()`, or similar) should support an `info()` method, which returns an `email.message.Message` instance.

The *request* object (usually a `urllib.request.Request` instance) must support the method `get_full_url()` and the attributes `host`, `unverifiable` and `origin_req_host`, as documented by `urllib.request`. The request is used to set default values for cookie-attributes as well as for checking that the cookie is allowed to be set.

Modifié dans la version 3.3 : *request* object needs `origin_req_host` attribute. Dependency on a deprecated method `get_origin_req_host()` has been removed.

`CookieJar.`**`set_policy`**(*policy*)

Set the `CookiePolicy` instance to be used.

`CookieJar.make_cookies(response, request)`

Return sequence of `Cookie` objects extracted from *response* object.

See the documentation for `extract_cookies()` for the interfaces required of the *response* and *request* arguments.

`CookieJar.set_cookie_if_ok(cookie, request)`

Set a `Cookie` if policy says it's OK to do so.

`CookieJar.set_cookie(cookie)`

Set a `Cookie`, without checking with policy to see whether or not it should be set.

`CookieJar.clear([domain[, path[, name]]])`

Clear some cookies.

If invoked without arguments, clear all cookies. If given a single argument, only cookies belonging to that *domain* will be removed. If given two arguments, cookies belonging to the specified *domain* and URL *path* are removed. If given three arguments, then the cookie with the specified *domain*, *path* and *name* is removed.

Raises `KeyError` if no matching cookie exists.

`CookieJar.clear_session_cookies()`

Discard all session cookies.

Discards all contained cookies that have a true `discard` attribute (usually because they had either no `max-age` or `expires` cookie-attribute, or an explicit `discard` cookie-attribute). For interactive browsers, the end of a session usually corresponds to closing the browser window.

Note that the `save()` method won't save session cookies anyway, unless you ask otherwise by passing a true *ignore_discard* argument.

`FileCookieJar` implements the following additional methods :

`FileCookieJar.save(filename=None, ignore_discard=False, ignore_expires=False)`

Save cookies to a file.

This base class raises `NotImplementedError`. Subclasses may leave this method unimplemented.

filename is the name of file in which to save cookies. If *filename* is not specified, `self.filename` is used (whose default is the value passed to the constructor, if any) ; if `self.filename` is `None`, `ValueError` is raised.

ignore_discard : save even cookies set to be discarded. *ignore_expires* : save even cookies that have expired

The file is overwritten if it already exists, thus wiping all the cookies it contains. Saved cookies can be restored later using the `load()` or `revert()` methods.

`FileCookieJar.load(filename=None, ignore_discard=False, ignore_expires=False)`

Load cookies from a file.

Old cookies are kept unless overwritten by newly loaded ones.

Arguments are as for `save()`.

The named file must be in the format understood by the class, or `LoadError` will be raised. Also, `OSError` may be raised, for example if the file does not exist.

Modifié dans la version 3.3 : `IOError` était normalement levée, elle est maintenant un alias de `OSError`.

`FileCookieJar.revert(filename=None, ignore_discard=False, ignore_expires=False)`

Clear all cookies and reload cookies from a saved file.

`revert()` can raise the same exceptions as `load()`. If there is a failure, the object's state will not be altered.

`FileCookieJar` instances have the following public attributes :

`FileCookieJar.filename`

Filename of default file in which to keep cookies. This attribute may be assigned to.

`FileCookieJar.delayload`

If true, load cookies lazily from disk. This attribute should not be assigned to. This is only a hint, since this only affects performance, not behaviour (unless the cookies on disk are changing). A `CookieJar` object may ignore it. None of the `FileCookieJar` classes included in the standard library lazily loads cookies.

22.19.2 FileCookieJar subclasses and co-operation with web browsers

The following `CookieJar` subclasses are provided for reading and writing.

class `http.cookiejar.MozillaCookieJar`(*filename=None*, *delayload=None*, *policy=None*)

A `FileCookieJar` that can load from and save cookies to disk in the Mozilla `cookies.txt` file format (which is also used by curl and the Lynx and Netscape browsers).

> **Note**
>
> This loses information about **RFC 2965** cookies, and also about newer or non-standard cookie-attributes such as `port`.

> **Avertissement**
>
> Back up your cookies before saving if you have cookies whose loss / corruption would be inconvenient (there are some subtleties which may lead to slight changes in the file over a load / save round-trip).

Also note that cookies saved while Mozilla is running will get clobbered by Mozilla.

class `http.cookiejar.LWPCookieJar`(*filename=None*, *delayload=None*, *policy=None*)

A `FileCookieJar` that can load from and save cookies to disk in format compatible with the libwww-perl library's `Set-Cookie3` file format. This is convenient if you want to store cookies in a human-readable file.

Modifié dans la version 3.8 : The filename parameter supports a *path-like object*.

22.19.3 CookiePolicy Objects

Objects implementing the `CookiePolicy` interface have the following methods :

`CookiePolicy.set_ok`(*cookie*, *request*)

Return boolean value indicating whether cookie should be accepted from server.

cookie is a `Cookie` instance. *request* is an object implementing the interface defined by the documentation for `CookieJar.extract_cookies()`.

`CookiePolicy.return_ok`(*cookie*, *request*)

Return boolean value indicating whether cookie should be returned to server.

cookie is a `Cookie` instance. *request* is an object implementing the interface defined by the documentation for `CookieJar.add_cookie_header()`.

`CookiePolicy.domain_return_ok`(*domain*, *request*)

Return `False` if cookies should not be returned, given cookie domain.

This method is an optimization. It removes the need for checking every cookie with a particular domain (which might involve reading many files). Returning true from `domain_return_ok()` and `path_return_ok()` leaves all the work to `return_ok()`.

If `domain_return_ok()` returns true for the cookie domain, `path_return_ok()` is called for the cookie path. Otherwise, `path_return_ok()` and `return_ok()` are never called for that cookie domain. If `path_return_ok()` returns true, `return_ok()` is called with the `Cookie` object itself for a full check. Otherwise, `return_ok()` is never called for that cookie path.

Note that `domain_return_ok()` is called for every *cookie* domain, not just for the *request* domain. For example, the function might be called with both `".example.com"` and `"www.example.com"` if the request domain is `"www.example.com"`. The same goes for `path_return_ok()`.

The *request* argument is as documented for `return_ok()`.

`CookiePolicy.`**`path_return_ok`**(*path*, *request*)

Return `False` if cookies should not be returned, given cookie path.

See the documentation for `domain_return_ok()`.

In addition to implementing the methods above, implementations of the `CookiePolicy` interface must also supply the following attributes, indicating which protocols should be used, and how. All of these attributes may be assigned to.

`CookiePolicy.`**`netscape`**

Implement Netscape protocol.

`CookiePolicy.`**`rfc2965`**

Implement **RFC 2965** protocol.

`CookiePolicy.`**`hide_cookie2`**

Don't add `Cookie2` header to requests (the presence of this header indicates to the server that we understand **RFC 2965** cookies).

The most useful way to define a `CookiePolicy` class is by subclassing from `DefaultCookiePolicy` and overriding some or all of the methods above. `CookiePolicy` itself may be used as a 'null policy' to allow setting and receiving any and all cookies (this is unlikely to be useful).

22.19.4 DefaultCookiePolicy Objects

Implements the standard rules for accepting and returning cookies.

Both **RFC 2965** and Netscape cookies are covered. RFC 2965 handling is switched off by default.

The easiest way to provide your own policy is to override this class and call its methods in your overridden implementations before adding your own additional checks :

```
import http.cookiejar
class MyCookiePolicy(http.cookiejar.DefaultCookiePolicy):
    def set_ok(self, cookie, request):
        if not http.cookiejar.DefaultCookiePolicy.set_ok(self, cookie, request):
            return False
        if i_dont_want_to_store_this_cookie(cookie):
            return False
        return True
```

In addition to the features required to implement the `CookiePolicy` interface, this class allows you to block and allow domains from setting and receiving cookies. There are also some strictness switches that allow you to tighten up the rather loose Netscape protocol rules a little bit (at the cost of blocking some benign cookies).

A domain blocklist and allowlist is provided (both off by default). Only domains not in the blocklist and present in the allowlist (if the allowlist is active) participate in cookie setting and returning. Use the *blocked_domains* constructor argument, and `blocked_domains()` and `set_blocked_domains()` methods (and the corresponding argument and methods for *allowed_domains*). If you set an allowlist, you can turn it off again by setting it to `None`.

Domains in block or allow lists that do not start with a dot must equal the cookie domain to be matched. For example, `"example.com"` matches a blocklist entry of `"example.com"`, but `"www.example.com"` does not. Domains that do start with a dot are matched by more specific domains too. For example, both `"www.example.com"` and `"www.coyote.example.com"` match `".example.com"` (but `"example.com"` itself does not). IP addresses are an exception, and must match exactly. For example, if blocked_domains contains `"192.168.1.2"` and `".168.1.2"`, 192.168.1.2 is blocked, but 193.168.1.2 is not.

`DefaultCookiePolicy` implements the following additional methods :

`DefaultCookiePolicy.blocked_domains()`

Return the sequence of blocked domains (as a tuple).

`DefaultCookiePolicy.set_blocked_domains(blocked_domains)`

Set the sequence of blocked domains.

`DefaultCookiePolicy.is_blocked(domain)`

Return `True` if *domain* is on the blocklist for setting or receiving cookies.

`DefaultCookiePolicy.allowed_domains()`

Return `None`, or the sequence of allowed domains (as a tuple).

`DefaultCookiePolicy.set_allowed_domains(allowed_domains)`

Set the sequence of allowed domains, or `None`.

`DefaultCookiePolicy.is_not_allowed(domain)`

Return `True` if *domain* is not on the allowlist for setting or receiving cookies.

`DefaultCookiePolicy` instances have the following attributes, which are all initialised from the constructor arguments of the same name, and which may all be assigned to.

`DefaultCookiePolicy.rfc2109_as_netscape`

If true, request that the `CookieJar` instance downgrade **RFC 2109** cookies (ie. cookies received in a `Set-Cookie` header with a version cookie-attribute of 1) to Netscape cookies by setting the version attribute of the `Cookie` instance to 0. The default value is `None`, in which case RFC 2109 cookies are downgraded if and only if **RFC 2965** handling is turned off. Therefore, RFC 2109 cookies are downgraded by default.

General strictness switches :

`DefaultCookiePolicy.strict_domain`

Don't allow sites to set two-component domains with country-code top-level domains like `.co.uk`, `.gov.uk`, `.co.nz`.etc. This is far from perfect and isn't guaranteed to work !

RFC 2965 protocol strictness switches :

`DefaultCookiePolicy.strict_rfc2965_unverifiable`

Follow **RFC 2965** rules on unverifiable transactions (usually, an unverifiable transaction is one resulting from a redirect or a request for an image hosted on another site). If this is false, cookies are *never* blocked on the basis of verifiability

Netscape protocol strictness switches :

`DefaultCookiePolicy.strict_ns_unverifiable`

Apply **RFC 2965** rules on unverifiable transactions even to Netscape cookies.

`DefaultCookiePolicy.strict_ns_domain`

Flags indicating how strict to be with domain-matching rules for Netscape cookies. See below for acceptable values.

`DefaultCookiePolicy.strict_ns_set_initial_dollar`

Ignore cookies in Set-Cookie : headers that have names starting with `'$'`.

`DefaultCookiePolicy.strict_ns_set_path`

Don't allow setting cookies whose path doesn't path-match request URI.

`strict_ns_domain` is a collection of flags. Its value is constructed by or-ing together (for example, `DomainStrictNoDots|DomainStrictNonDomain` means both flags are set).

`DefaultCookiePolicy.DomainStrictNoDots`

When setting cookies, the 'host prefix' must not contain a dot (eg. `www.foo.bar.com` can't set a cookie for `.bar.com`, because `www.foo` contains a dot).

`DefaultCookiePolicy.`**`DomainStrictNonDomain`**

Cookies that did not explicitly specify a `domain` cookie-attribute can only be returned to a domain equal to the domain that set the cookie (eg. `spam.example.com` won't be returned cookies from `example.com` that had no `domain` cookie-attribute).

`DefaultCookiePolicy.`**`DomainRFC2965Match`**

When setting cookies, require a full **RFC 2965** domain-match.

The following attributes are provided for convenience, and are the most useful combinations of the above flags :

`DefaultCookiePolicy.`**`DomainLiberal`**

Equivalent to 0 (ie. all of the above Netscape domain strictness flags switched off).

`DefaultCookiePolicy.`**`DomainStrict`**

Equivalent to `DomainStrictNoDots|DomainStrictNonDomain`.

22.19.5 Objets *Cookie*

`Cookie` instances have Python attributes roughly corresponding to the standard cookie-attributes specified in the various cookie standards. The correspondence is not one-to-one, because there are complicated rules for assigning default values, because the `max-age` and `expires` cookie-attributes contain equivalent information, and because **RFC 2109** cookies may be 'downgraded' by *`http.cookiejar`* from version 1 to version 0 (Netscape) cookies.

Assignment to these attributes should not be necessary other than in rare circumstances in a *`CookiePolicy`* method. The class does not enforce internal consistency, so you should know what you're doing if you do that.

`Cookie.`**`version`**

Integer or *`None`*. Netscape cookies have *`version`* 0. **RFC 2965** and **RFC 2109** cookies have a `version` cookie-attribute of 1. However, note that *`http.cookiejar`* may 'downgrade' RFC 2109 cookies to Netscape cookies, in which case *`version`* is 0.

`Cookie.`**`name`**

Cookie name (a string).

`Cookie.`**`value`**

Cookie value (a string), or *`None`*.

`Cookie.`**`port`**

String representing a port or a set of ports (eg. '80', or '80,8080'), or *`None`*.

`Cookie.`**`domain`**

Cookie domain (a string).

`Cookie.`**`path`**

Cookie path (a string, eg. `'/acme/rocket_launchers'`).

`Cookie.`**`secure`**

`True` if cookie should only be returned over a secure connection.

`Cookie.`**`expires`**

Integer expiry date in seconds since epoch, or *`None`*. See also the *`is_expired()`* method.

`Cookie.`**`discard`**

`True` if this is a session cookie.

`Cookie.`**`comment`**

String comment from the server explaining the function of this cookie, or *`None`*.

`Cookie.`**`comment_url`**

URL linking to a comment from the server explaining the function of this cookie, or *`None`*.

`Cookie.rfc2109`

`True` if this cookie was received as an **RFC 2109** cookie (ie. the cookie arrived in a `Set-Cookie` header, and the value of the Version cookie-attribute in that header was 1). This attribute is provided because `http.cookiejar` may 'downgrade' RFC 2109 cookies to Netscape cookies, in which case `version` is 0.

`Cookie.port_specified`

`True` if a port or set of ports was explicitly specified by the server (in the `Set-Cookie` / `Set-Cookie2` header).

`Cookie.domain_specified`

`True` if a domain was explicitly specified by the server.

`Cookie.domain_initial_dot`

`True` if the domain explicitly specified by the server began with a dot (`'.'`).

Cookies may have additional non-standard cookie-attributes. These may be accessed using the following methods :

`Cookie.has_nonstandard_attr`(*name*)

Return `True` if cookie has the named cookie-attribute.

`Cookie.get_nonstandard_attr`(*name*, *default=None*)

If cookie has the named cookie-attribute, return its value. Otherwise, return *default*.

`Cookie.set_nonstandard_attr`(*name*, *value*)

Set the value of the named cookie-attribute.

The `Cookie` class also defines the following method :

`Cookie.is_expired`(*now=None*)

`True` if cookie has passed the time at which the server requested it should expire. If *now* is given (in seconds since the epoch), return whether the cookie has expired at the specified time.

22.19.6 Exemples

The first example shows the most common usage of `http.cookiejar` :

```
import http.cookiejar, urllib.request
cj = http.cookiejar.CookieJar()
opener = urllib.request.build_opener(urllib.request.HTTPCookieProcessor(cj))
r = opener.open("http://example.com/")
```

This example illustrates how to open a URL using your Netscape, Mozilla, or Lynx cookies (assumes Unix/Netscape convention for location of the cookies file) :

```
import os, http.cookiejar, urllib.request
cj = http.cookiejar.MozillaCookieJar()
cj.load(os.path.join(os.path.expanduser("~"), ".netscape", "cookies.txt"))
opener = urllib.request.build_opener(urllib.request.HTTPCookieProcessor(cj))
r = opener.open("http://example.com/")
```

The next example illustrates the use of `DefaultCookiePolicy`. Turn on **RFC 2965** cookies, be more strict about domains when setting and returning Netscape cookies, and block some domains from setting cookies or having them returned :

```
import urllib.request
from http.cookiejar import CookieJar, DefaultCookiePolicy
policy = DefaultCookiePolicy(
    rfc2965=True, strict_ns_domain=Policy.DomainStrict,
    blocked_domains=["ads.net", ".ads.net"])
cj = CookieJar(policy)
```

(suite sur la page suivante)

(suite de la page précédente)

```
opener = urllib.request.build_opener(urllib.request.HTTPCookieProcessor(cj))
r = opener.open("http://example.com/")
```

22.20 xmlrpc --- XMLRPC server and client modules

XML-RPC est une méthode pour appeler des procédures distantes utilisant XML via HTTP. XML-RPC permet à un client d'appeler des fonctions avec leurs arguments sur un serveur distant (désigné par une URI), et recevoir en retour des données structurées.

`xmlrpc` est un paquet rassemblant un client et un serveur XML-RPC. Ces modules sont :

- *xmlrpc.client*
- *xmlrpc.server*

22.21 xmlrpc.client --- XML-RPC client access

Source code : Lib/xmlrpc/client.py

XML-RPC is a Remote Procedure Call method that uses XML passed via HTTP(S) as a transport. With it, a client can call methods with parameters on a remote server (the server is named by a URI) and get back structured data. This module supports writing XML-RPC client code ; it handles all the details of translating between conformable Python objects and XML on the wire.

> **Avertissement**
>
> The *xmlrpc.client* module is not secure against maliciously constructed data. If you need to parse untrusted or unauthenticated data see *Vulnérabilités XML*.

Modifié dans la version 3.5 : For HTTPS URIs, *xmlrpc.client* now performs all the necessary certificate and hostname checks by default.

Availability : not WASI.

This module does not work or is not available on WebAssembly. See *Plateformes WebAssembly* for more information.

class xmlrpc.client.**ServerProxy**(*uri*, *transport=None*, *encoding=None*, *verbose=False*, *allow_none=False*, *use_datetime=False*, *use_builtin_types=False*, ***, *headers=()*, *context=None*)

A *ServerProxy* instance is an object that manages communication with a remote XML-RPC server. The required first argument is a URI (Uniform Resource Indicator), and will normally be the URL of the server. The optional second argument is a transport factory instance ; by default it is an internal `SafeTransport` instance for https : URLs and an internal HTTP `Transport` instance otherwise. The optional third argument is an encoding, by default UTF-8. The optional fourth argument is a debugging flag.

The following parameters govern the use of the returned proxy instance. If *allow_none* is true, the Python constant `None` will be translated into XML ; the default behaviour is for `None` to raise a *TypeError*. This is a commonly used extension to the XML-RPC specification, but isn't supported by all clients and servers ; see http ://ontosys.com/xml-rpc/extensions.php for a description. The *use_builtin_types* flag can be used to cause date/time values to be presented as *datetime.datetime* objects and binary data to be presented as *bytes* objects ; this flag is false by default. *datetime.datetime*, *bytes* and *bytearray* objects may be passed to calls. The *headers* parameter is an optional sequence of HTTP headers to send with each request, expressed as a sequence of 2-tuples representing the header name and value. (e.g. `[('Header-Name', 'value')]`). The obsolete *use_datetime* flag is similar to *use_builtin_types* but it applies only to date/time values.

Modifié dans la version 3.3 : The *use_builtin_types* flag was added.

Modifié dans la version 3.8 : The *headers* parameter was added.

Both the HTTP and HTTPS transports support the URL syntax extension for HTTP Basic Authentication : `http://user:pass@host:port/path`. The `user:pass` portion will be base64-encoded as an HTTP 'Authorization' header, and sent to the remote server as part of the connection process when invoking an XML-RPC method. You only need to use this if the remote server requires a Basic Authentication user and password. If an HTTPS URL is provided, *context* may be `ssl.SSLContext` and configures the SSL settings of the underlying HTTPS connection.

The returned instance is a proxy object with methods that can be used to invoke corresponding RPC calls on the remote server. If the remote server supports the introspection API, the proxy can also be used to query the remote server for the methods it supports (service discovery) and fetch other server-associated metadata.

Types that are conformable (e.g. that can be marshalled through XML), include the following (and except where noted, they are unmarshalled as the same Python type) :

XML-RPC type	Type Python
`boolean`	`bool`
`int`, `i1`, `i2`, `i4`, `i8` or `biginteger`	`int` in range from -2147483648 to 2147483647. Values get the `<int>` tag.
`double` or `float`	`float`. Values get the `<double>` tag.
`string`	`str`
`array`	`list` or `tuple` containing conformable elements. Arrays are returned as `lists`.
`struct`	`dict`. Keys must be strings, values may be any conformable type. Objects of user-defined classes can be passed in ; only their `__dict__` attribute is transmitted.
`dateTime.iso8601`	`DateTime` or `datetime.datetime`. Returned type depends on values of *use_builtin_types* and *use_datetime* flags.
`base64`	`Binary`, `bytes` or `bytearray`. Returned type depends on the value of the *use_builtin_types* flag.
`nil`	The `None` constant. Passing is allowed only if *allow_none* is true.
`bigdecimal`	`decimal.Decimal`. Returned type only.

This is the full set of data types supported by XML-RPC. Method calls may also raise a special `Fault` instance, used to signal XML-RPC server errors, or `ProtocolError` used to signal an error in the HTTP/HTTPS transport layer. Both `Fault` and `ProtocolError` derive from a base class called `Error`. Note that the xmlrpc client module currently does not marshal instances of subclasses of built-in types.

When passing strings, characters special to XML such as `<`, `>`, and `&` will be automatically escaped. However, it's the caller's responsibility to ensure that the string is free of characters that aren't allowed in XML, such as the control characters with ASCII values between 0 and 31 (except, of course, tab, newline and carriage return) ; failing to do this will result in an XML-RPC request that isn't well-formed XML. If you have to pass arbitrary bytes via XML-RPC, use `bytes` or `bytearray` classes or the `Binary` wrapper class described below.

`Server` is retained as an alias for `ServerProxy` for backwards compatibility. New code should use `ServerProxy`.

Modifié dans la version 3.5 : Added the *context* argument.

Modifié dans la version 3.6 : Added support of type tags with prefixes (e.g. `ex:nil`). Added support of unmarshalling additional types used by Apache XML-RPC implementation for numerics : `i1`, `i2`, `i8`, `biginteger`, `float` and `bigdecimal`. See https://ws.apache.org/xmlrpc/types.html for a description.

Voir aussi

XML-RPC HOWTO
A good description of XML-RPC operation and client software in several languages. Contains pretty much everything an XML-RPC client developer needs to know.

XML-RPC Introspection

Describes the XML-RPC protocol extension for introspection.

XML-RPC Specification
The official specification.

22.21.1 ServerProxy Objects

A `ServerProxy` instance has a method corresponding to each remote procedure call accepted by the XML-RPC server. Calling the method performs an RPC, dispatched by both name and argument signature (e.g. the same method name can be overloaded with multiple argument signatures). The RPC finishes by returning a value, which may be either returned data in a conformant type or a `Fault` or `ProtocolError` object indicating an error.

Servers that support the XML introspection API support some common methods grouped under the reserved `system` attribute :

`ServerProxy.system.listMethods()`
This method returns a list of strings, one for each (non-system) method supported by the XML-RPC server.

`ServerProxy.system.methodSignature(name)`
This method takes one parameter, the name of a method implemented by the XML-RPC server. It returns an array of possible signatures for this method. A signature is an array of types. The first of these types is the return type of the method, the rest are parameters.

Because multiple signatures (ie. overloading) is permitted, this method returns a list of signatures rather than a singleton.

Signatures themselves are restricted to the top level parameters expected by a method. For instance if a method expects one array of structs as a parameter, and it returns a string, its signature is simply "string, array". If it expects three integers and returns a string, its signature is "string, int, int, int".

If no signature is defined for the method, a non-array value is returned. In Python this means that the type of the returned value will be something other than list.

`ServerProxy.system.methodHelp(name)`
This method takes one parameter, the name of a method implemented by the XML-RPC server. It returns a documentation string describing the use of that method. If no such string is available, an empty string is returned. The documentation string may contain HTML markup.

Modifié dans la version 3.5 : Instances of `ServerProxy` support the *context manager* protocol for closing the underlying transport.

A working example follows. The server code :

```
from xmlrpc.server import SimpleXMLRPCServer

def is_even(n):
    return n % 2 == 0

server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_function(is_even, "is_even")
server.serve_forever()
```

The client code for the preceding server :

```
import xmlrpc.client

with xmlrpc.client.ServerProxy("http://localhost:8000/") as proxy:
    print("3 is even: %s" % str(proxy.is_even(3)))
    print("100 is even: %s" % str(proxy.is_even(100)))
```

22.21.2 Objets DateTime

class xmlrpc.client.**DateTime**

This class may be initialized with seconds since the epoch, a time tuple, an ISO 8601 time/date string, or a *datetime.datetime* instance. It has the following methods, supported mainly for internal use by the marshalling/unmarshalling code :

decode(*string*)

Accept a string as the instance's new time value.

encode(*out*)

Write the XML-RPC encoding of this *DateTime* item to the *out* stream object.

It also supports certain of Python's built-in operators through `rich comparison` and `__repr__()` methods.

A working example follows. The server code :

```
import datetime
from xmlrpc.server import SimpleXMLRPCServer
import xmlrpc.client

def today():
    today = datetime.datetime.today()
    return xmlrpc.client.DateTime(today)

server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_function(today, "today")
server.serve_forever()
```

The client code for the preceding server :

```
import xmlrpc.client
import datetime

proxy = xmlrpc.client.ServerProxy("http://localhost:8000/")

today = proxy.today()
# convert the ISO8601 string to a datetime object
converted = datetime.datetime.strptime(today.value, "%Y%m%dT%H:%M:%S")
print("Today: %s" % converted.strftime("%d.%m.%Y, %H:%M"))
```

22.21.3 Binary Objects

class xmlrpc.client.**Binary**

This class may be initialized from bytes data (which may include NULs). The primary access to the content of a *Binary* object is provided by an attribute :

data

The binary data encapsulated by the *Binary* instance. The data is provided as a *bytes* object.

Binary objects have the following methods, supported mainly for internal use by the marshalling/unmarshalling code :

decode(*bytes*)

Accept a base64 *bytes* object and decode it as the instance's new data.

encode(*out*)

Write the XML-RPC base 64 encoding of this binary item to the *out* stream object.

The encoded data will have newlines every 76 characters as per **RFC 2045 section 6.8**, which was the de facto standard base64 specification when the XML-RPC spec was written.

It also supports certain of Python's built-in operators through `__eq__()` and `__ne__()` methods.

Example usage of the binary objects. We're going to transfer an image over XMLRPC :

```
from xmlrpc.server import SimpleXMLRPCServer
import xmlrpc.client

def python_logo():
    with open("python_logo.jpg", "rb") as handle:
        return xmlrpc.client.Binary(handle.read())

server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_function(python_logo, 'python_logo')

server.serve_forever()
```

The client gets the image and saves it to a file :

```
import xmlrpc.client

proxy = xmlrpc.client.ServerProxy("http://localhost:8000/")
with open("fetched_python_logo.jpg", "wb") as handle:
    handle.write(proxy.python_logo().data)
```

22.21.4 Fault Objects

class xmlrpc.client.**Fault**

A *Fault* object encapsulates the content of an XML-RPC fault tag. Fault objects have the following attributes :

faultCode

An int indicating the fault type.

faultString

A string containing a diagnostic message associated with the fault.

In the following example we're going to intentionally cause a *Fault* by returning a complex type object. The server code :

```
from xmlrpc.server import SimpleXMLRPCServer

# A marshalling error is going to occur because we're returning a
# complex number
def add(x, y):
    return x+y+0j

server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_function(add, 'add')

server.serve_forever()
```

The client code for the preceding server :

```
import xmlrpc.client

proxy = xmlrpc.client.ServerProxy("http://localhost:8000/")
try:
```

(suite sur la page suivante)

(suite de la page précédente)

```
    proxy.add(2, 5)
except xmlrpc.client.Fault as err:
    print("A fault occurred")
    print("Fault code: %d" % err.faultCode)
    print("Fault string: %s" % err.faultString)
```

22.21.5 ProtocolError Objects

class xmlrpc.client.**ProtocolError**

A *ProtocolError* object describes a protocol error in the underlying transport layer (such as a 404 'not found' error if the server named by the URI does not exist). It has the following attributes :

url

The URI or URL that triggered the error.

errcode

The error code.

errmsg

The error message or diagnostic string.

headers

A dict containing the headers of the HTTP/HTTPS request that triggered the error.

In the following example we're going to intentionally cause a *ProtocolError* by providing an invalid URI :

```
import xmlrpc.client

# create a ServerProxy with a URI that doesn't respond to XMLRPC requests
proxy = xmlrpc.client.ServerProxy("http://google.com/")

try:
    proxy.some_method()
except xmlrpc.client.ProtocolError as err:
    print("A protocol error occurred")
    print("URL: %s" % err.url)
    print("HTTP/HTTPS headers: %s" % err.headers)
    print("Error code: %d" % err.errcode)
    print("Error message: %s" % err.errmsg)
```

22.21.6 MultiCall Objects

The *MultiCall* object provides a way to encapsulate multiple calls to a remote server into a single request [1].

class xmlrpc.client.**MultiCall**(*server*)

Create an object used to boxcar method calls. *server* is the eventual target of the call. Calls can be made to the result object, but they will immediately return `None`, and only store the call name and parameters in the *MultiCall* object. Calling the object itself causes all stored calls to be transmitted as a single `system.multicall` request. The result of this call is a *generator*; iterating over this generator yields the individual results.

A usage example of this class follows. The server code :

```
from xmlrpc.server import SimpleXMLRPCServer

def add(x, y):
```

(suite sur la page suivante)

1. This approach has been first presented in a discussion on xmlrpc.com.

(suite de la page précédente)

```
    return x + y

def subtract(x, y):
    return x - y

def multiply(x, y):
    return x * y

def divide(x, y):
    return x // y

# A simple server with simple arithmetic functions
server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_multicall_functions()
server.register_function(add, 'add')
server.register_function(subtract, 'subtract')
server.register_function(multiply, 'multiply')
server.register_function(divide, 'divide')
server.serve_forever()
```

The client code for the preceding server :

```
import xmlrpc.client

proxy = xmlrpc.client.ServerProxy("http://localhost:8000/")
multicall = xmlrpc.client.MultiCall(proxy)
multicall.add(7, 3)
multicall.subtract(7, 3)
multicall.multiply(7, 3)
multicall.divide(7, 3)
result = multicall()

print("7+3=%d, 7-3=%d, 7*3=%d, 7//3=%d" % tuple(result))
```

22.21.7 Convenience Functions

`xmlrpc.client.`**`dumps`**(*params*, *methodname=None*, *methodresponse=None*, *encoding=None*, *allow_none=False*)

Convert *params* into an XML-RPC request. or into a response if *methodresponse* is true. *params* can be either a tuple of arguments or an instance of the `Fault` exception class. If *methodresponse* is true, only a single value can be returned, meaning that *params* must be of length 1. *encoding*, if supplied, is the encoding to use in the generated XML ; the default is UTF-8. Python's `None` value cannot be used in standard XML-RPC ; to allow using it via an extension, provide a true value for *allow_none*.

`xmlrpc.client.`**`loads`**(*data*, *use_datetime=False*, *use_builtin_types=False*)

Convert an XML-RPC request or response into Python objects, a `(params, methodname)`. *params* is a tuple of argument ; *methodname* is a string, or `None` if no method name is present in the packet. If the XML-RPC packet represents a fault condition, this function will raise a `Fault` exception. The *use_builtin_types* flag can be used to cause date/time values to be presented as `datetime.datetime` objects and binary data to be presented as `bytes` objects ; this flag is false by default.

The obsolete *use_datetime* flag is similar to *use_builtin_types* but it applies only to date/time values.

Modifié dans la version 3.3 : The *use_builtin_types* flag was added.

22.21.8 Example of Client Usage

```
# simple test program (from the XML-RPC specification)
from xmlrpc.client import ServerProxy, Error

# server = ServerProxy("http://localhost:8000") # local server
with ServerProxy("http://betty.userland.com") as proxy:

    print(proxy)

    try:
        print(proxy.examples.getStateName(41))
    except Error as v:
        print("ERROR", v)
```

To access an XML-RPC server through a HTTP proxy, you need to define a custom transport. The following example shows how :

```
import http.client
import xmlrpc.client

class ProxiedTransport(xmlrpc.client.Transport):

    def set_proxy(self, host, port=None, headers=None):
        self.proxy = host, port
        self.proxy_headers = headers

    def make_connection(self, host):
        connection = http.client.HTTPConnection(*self.proxy)
        connection.set_tunnel(host, headers=self.proxy_headers)
        self._connection = host, connection
        return connection

transport = ProxiedTransport()
transport.set_proxy('proxy-server', 8080)
server = xmlrpc.client.ServerProxy('http://betty.userland.com',␣
↪transport=transport)
print(server.examples.getStateName(41))
```

22.21.9 Example of Client and Server Usage

See *SimpleXMLRPCServer Example*.

Notes

22.22 `xmlrpc.server` --- Basic XML-RPC servers

Source code : Lib/xmlrpc/server.py

The `xmlrpc.server` module provides a basic server framework for XML-RPC servers written in Python. Servers can either be free standing, using `SimpleXMLRPCServer`, or embedded in a CGI environment, using `CGIXMLRPCRequestHandler`.

Avertissement

> The `xmlrpc.server` module is not secure against maliciously constructed data. If you need to parse untrusted or unauthenticated data see *Vulnérabilités XML*.

Availability : not WASI.

This module does not work or is not available on WebAssembly. See *Plateformes WebAssembly* for more information.

class xmlrpc.server.**SimpleXMLRPCServer**(*addr*, *requestHandler=SimpleXMLRPCRequestHandler*, *logRequests=True*, *allow_none=False*, *encoding=None*, *bind_and_activate=True*, *use_builtin_types=False*)

Create a new server instance. This class provides methods for registration of functions that can be called by the XML-RPC protocol. The *requestHandler* parameter should be a factory for request handler instances ; it defaults to `SimpleXMLRPCRequestHandler`. The *addr* and *requestHandler* parameters are passed to the `socketserver.TCPServer` constructor. If *logRequests* is true (the default), requests will be logged ; setting this parameter to false will turn off logging. The *allow_none* and *encoding* parameters are passed on to `xmlrpc.client` and control the XML-RPC responses that will be returned from the server. The *bind_and_activate* parameter controls whether `server_bind()` and `server_activate()` are called immediately by the constructor ; it defaults to true. Setting it to false allows code to manipulate the *allow_reuse_address* class variable before the address is bound. The *use_builtin_types* parameter is passed to the `loads()` function and controls which types are processed when date/times values or binary data are received ; it defaults to false.

Modifié dans la version 3.3 : The *use_builtin_types* flag was added.

class xmlrpc.server.**CGIXMLRPCRequestHandler**(*allow_none=False*, *encoding=None*, *use_builtin_types=False*)

Create a new instance to handle XML-RPC requests in a CGI environment. The *allow_none* and *encoding* parameters are passed on to `xmlrpc.client` and control the XML-RPC responses that will be returned from the server. The *use_builtin_types* parameter is passed to the `loads()` function and controls which types are processed when date/times values or binary data are received ; it defaults to false.

Modifié dans la version 3.3 : The *use_builtin_types* flag was added.

class xmlrpc.server.**SimpleXMLRPCRequestHandler**

Create a new request handler instance. This request handler supports `POST` requests and modifies logging so that the *logRequests* parameter to the `SimpleXMLRPCServer` constructor parameter is honored.

22.22.1 SimpleXMLRPCServer Objects

The `SimpleXMLRPCServer` class is based on `socketserver.TCPServer` and provides a means of creating simple, stand alone XML-RPC servers.

SimpleXMLRPCServer.**register_function**(*function=None*, *name=None*)

Register a function that can respond to XML-RPC requests. If *name* is given, it will be the method name associated with *function*, otherwise `function.__name__` will be used. *name* is a string, and may contain characters not legal in Python identifiers, including the period character.

This method can also be used as a decorator. When used as a decorator, *name* can only be given as a keyword argument to register *function* under *name*. If no *name* is given, `function.__name__` will be used.

Modifié dans la version 3.7 : `register_function()` can be used as a decorator.

SimpleXMLRPCServer.**register_instance**(*instance*, *allow_dotted_names=False*)

Register an object which is used to expose method names which have not been registered using `register_function()`. If *instance* contains a `_dispatch()` method, it is called with the requested method name and the parameters from the request. Its API is `def _dispatch(self, method, params)` (note that *params* does not represent a variable argument list). If it calls an underlying function to perform its task, that function is called as `func(*params)`, expanding the parameter list. The return value from `_dispatch()` is returned to the client as the result. If *instance* does not have a `_dispatch()` method, it is searched for an attribute matching the name of the requested method.

If the optional *allow_dotted_names* argument is true and the instance does not have a `_dispatch()` method, then if the requested method name contains periods, each component of the method name is searched for

individually, with the effect that a simple hierarchical search is performed. The value found from this search is then called with the parameters from the request, and the return value is passed back to the client.

> **Avertissement**
>
> Enabling the *allow_dotted_names* option allows intruders to access your module's global variables and may allow intruders to execute arbitrary code on your machine. Only use this option on a secure, closed network.

`SimpleXMLRPCServer.register_introspection_functions()`

Registers the XML-RPC introspection functions `system.listMethods`, `system.methodHelp` and `system.methodSignature`.

`SimpleXMLRPCServer.register_multicall_functions()`

Registers the XML-RPC multicall function system.multicall.

`SimpleXMLRPCRequestHandler.rpc_paths`

An attribute value that must be a tuple listing valid path portions of the URL for receiving XML-RPC requests. Requests posted to other paths will result in a 404 "no such page" HTTP error. If this tuple is empty, all paths will be considered valid. The default value is `('/', '/RPC2')`.

SimpleXMLRPCServer Example

Server code :

```
from xmlrpc.server import SimpleXMLRPCServer
from xmlrpc.server import SimpleXMLRPCRequestHandler

# Restrict to a particular path.
class RequestHandler(SimpleXMLRPCRequestHandler):
    rpc_paths = ('/RPC2',)

# Create server
with SimpleXMLRPCServer(('localhost', 8000),
                        requestHandler=RequestHandler) as server:
    server.register_introspection_functions()

    # Register pow() function; this will use the value of
    # pow.__name__ as the name, which is just 'pow'.
    server.register_function(pow)

    # Register a function under a different name
    def adder_function(x, y):
        return x + y
    server.register_function(adder_function, 'add')

    # Register an instance; all the methods of the instance are
    # published as XML-RPC methods (in this case, just 'mul').
    class MyFuncs:
        def mul(self, x, y):
            return x * y

    server.register_instance(MyFuncs())

    # Run the server's main loop
    server.serve_forever()
```

The following client code will call the methods made available by the preceding server :

```
import xmlrpc.client

s = xmlrpc.client.ServerProxy('http://localhost:8000')
print(s.pow(2,3))  # Returns 2**3 = 8
print(s.add(2,3))  # Returns 5
print(s.mul(5,2))  # Returns 5*2 = 10

# Print list of available methods
print(s.system.listMethods())
```

`register_function()` can also be used as a decorator. The previous server example can register functions in a decorator way :

```
from xmlrpc.server import SimpleXMLRPCServer
from xmlrpc.server import SimpleXMLRPCRequestHandler

class RequestHandler(SimpleXMLRPCRequestHandler):
    rpc_paths = ('/RPC2',)

with SimpleXMLRPCServer(('localhost', 8000),
                        requestHandler=RequestHandler) as server:
    server.register_introspection_functions()

    # Register pow() function; this will use the value of
    # pow.__name__ as the name, which is just 'pow'.
    server.register_function(pow)

    # Register a function under a different name, using
    # register_function as a decorator. *name* can only be given
    # as a keyword argument.
    @server.register_function(name='add')
    def adder_function(x, y):
        return x + y

    # Register a function under function.__name__.
    @server.register_function
    def mul(x, y):
        return x * y

    server.serve_forever()
```

The following example included in the `Lib/xmlrpc/server.py` module shows a server allowing dotted names and registering a multicall function.

> **Avertissement**
>
> Enabling the *allow_dotted_names* option allows intruders to access your module's global variables and may allow intruders to execute arbitrary code on your machine. Only use this example only within a secure, closed network.

```
import datetime

class ExampleService:
    def getData(self):
        return '42'

    class currentTime:
```

(suite sur la page suivante)

(suite de la page précédente)

```
        @staticmethod
        def getCurrentTime():
            return datetime.datetime.now()

with SimpleXMLRPCServer(("localhost", 8000)) as server:
    server.register_function(pow)
    server.register_function(lambda x,y: x+y, 'add')
    server.register_instance(ExampleService(), allow_dotted_names=True)
    server.register_multicall_functions()
    print('Serving XML-RPC on localhost port 8000')
    try:
        server.serve_forever()
    except KeyboardInterrupt:
        print("\nKeyboard interrupt received, exiting.")
        sys.exit(0)
```

This ExampleService demo can be invoked from the command line :

```
python -m xmlrpc.server
```

The client that interacts with the above server is included in `Lib/xmlrpc/client.py` :

```
server = ServerProxy("http://localhost:8000")

try:
    print(server.currentTime.getCurrentTime())
except Error as v:
    print("ERROR", v)

multi = MultiCall(server)
multi.getData()
multi.pow(2,9)
multi.add(1,2)
try:
    for response in multi():
        print(response)
except Error as v:
    print("ERROR", v)
```

This client which interacts with the demo XMLRPC server can be invoked as :

```
python -m xmlrpc.client
```

22.22.2 CGIXMLRPCRequestHandler

The *`CGIXMLRPCRequestHandler`* class can be used to handle XML-RPC requests sent to Python CGI scripts.

`CGIXMLRPCRequestHandler.`**`register_function`**(*function=None*, *name=None*)

Register a function that can respond to XML-RPC requests. If *name* is given, it will be the method name associated with *function*, otherwise `function.__name__` will be used. *name* is a string, and may contain characters not legal in Python identifiers, including the period character.

This method can also be used as a decorator. When used as a decorator, *name* can only be given as a keyword argument to register *function* under *name*. If no *name* is given, `function.__name__` will be used.

Modifié dans la version 3.7 : *`register_function()`* can be used as a decorator.

`CGIXMLRPCRequestHandler.`**`register_instance`**(*instance*)

Register an object which is used to expose method names which have not been registered using

register_function(). If instance contains a `_dispatch()` method, it is called with the requested method name and the parameters from the request ; the return value is returned to the client as the result. If instance does not have a `_dispatch()` method, it is searched for an attribute matching the name of the requested method ; if the requested method name contains periods, each component of the method name is searched for individually, with the effect that a simple hierarchical search is performed. The value found from this search is then called with the parameters from the request, and the return value is passed back to the client.

`CGIXMLRPCRequestHandler.`**`register_introspection_functions`**`()`

Register the XML-RPC introspection functions `system.listMethods`, `system.methodHelp` and `system.methodSignature`.

`CGIXMLRPCRequestHandler.`**`register_multicall_functions`**`()`

Register the XML-RPC multicall function `system.multicall`.

`CGIXMLRPCRequestHandler.`**`handle_request`**`(`*request_text=None*`)`

Handle an XML-RPC request. If *request_text* is given, it should be the POST data provided by the HTTP server, otherwise the contents of stdin will be used.

Exemple :

```
class MyFuncs:
    def mul(self, x, y):
        return x * y


handler = CGIXMLRPCRequestHandler()
handler.register_function(pow)
handler.register_function(lambda x,y: x+y, 'add')
handler.register_introspection_functions()
handler.register_instance(MyFuncs())
handler.handle_request()
```

22.22.3 Documenting XMLRPC server

These classes extend the above classes to serve HTML documentation in response to HTTP GET requests. Servers can either be free standing, using *DocXMLRPCServer*, or embedded in a CGI environment, using *DocCGIXMLRPCRequestHandler*.

class `xmlrpc.server.`**`DocXMLRPCServer`**`(`*addr*, *requestHandler=DocXMLRPCRequestHandler*, *logRequests=True*, *allow_none=False*, *encoding=None*, *bind_and_activate=True*, *use_builtin_types=True*`)`

Create a new server instance. All parameters have the same meaning as for *SimpleXMLRPCServer* ; *requestHandler* defaults to *DocXMLRPCRequestHandler*.

Modifié dans la version 3.3 : The *use_builtin_types* flag was added.

class `xmlrpc.server.`**`DocCGIXMLRPCRequestHandler`**

Create a new instance to handle XML-RPC requests in a CGI environment.

class `xmlrpc.server.`**`DocXMLRPCRequestHandler`**

Create a new request handler instance. This request handler supports XML-RPC POST requests, documentation GET requests, and modifies logging so that the *logRequests* parameter to the *DocXMLRPCServer* constructor parameter is honored.

22.22.4 DocXMLRPCServer Objects

The *DocXMLRPCServer* class is derived from *SimpleXMLRPCServer* and provides a means of creating self-documenting, stand alone XML-RPC servers. HTTP POST requests are handled as XML-RPC method calls. HTTP GET requests are handled by generating pydoc-style HTML documentation. This allows a server to provide its own web-based documentation.

DocXMLRPCServer.**set_server_title**(*server_title*)

Set the title used in the generated HTML documentation. This title will be used inside the HTML "title" element.

DocXMLRPCServer.**set_server_name**(*server_name*)

Set the name used in the generated HTML documentation. This name will appear at the top of the generated documentation inside a "h1" element.

DocXMLRPCServer.**set_server_documentation**(*server_documentation*)

Set the description used in the generated HTML documentation. This description will appear as a paragraph, below the server name, in the documentation.

22.22.5 DocCGIXMLRPCRequestHandler

The `DocCGIXMLRPCRequestHandler` class is derived from `CGIXMLRPCRequestHandler` and provides a means of creating self-documenting, XML-RPC CGI scripts. HTTP POST requests are handled as XML-RPC method calls. HTTP GET requests are handled by generating pydoc-style HTML documentation. This allows a server to provide its own web-based documentation.

DocCGIXMLRPCRequestHandler.**set_server_title**(*server_title*)

Set the title used in the generated HTML documentation. This title will be used inside the HTML "title" element.

DocCGIXMLRPCRequestHandler.**set_server_name**(*server_name*)

Set the name used in the generated HTML documentation. This name will appear at the top of the generated documentation inside a "h1" element.

DocCGIXMLRPCRequestHandler.**set_server_documentation**(*server_documentation*)

Set the description used in the generated HTML documentation. This description will appear as a paragraph, below the server name, in the documentation.

22.23 `ipaddress` --- IPv4/IPv6 manipulation library

Code source : Lib/ipaddress.py

`ipaddress` propose des fonctionnalités pour créer, manipuler et opérer sur des réseaux et adresses IPv4 et IPv6.

Les fonctions et les classes dans ce module facilitent la gestion de différentes tâches reliée aux adresses IP, incluant vérifier si deux hôtes sont sur le même sous-réseau, itérer sur tous les hôtes d'un sous-réseau particulier, vérifier si une chaîne représente bien une adresse IP ou une définition de réseau valide, et ainsi de suite.

Ceci est la référence complète de l'API du module, pour un aperçu et introduction, voir ipaddress-howto.

Ajouté dans la version 3.3.

22.23.1 Fonctions fabriques pratiques

Le module `ipaddress` propose des fonctions fabriques pour facilement créer des adresses IP, réseaux et interfaces :

ipaddress.**ip_address**(*address*)

Return an `IPv4Address` or `IPv6Address` object depending on the IP address passed as argument. Either IPv4 or IPv6 addresses may be supplied ; integers less than `2**32` will be considered to be IPv4 by default. A `ValueError` is raised if *address* does not represent a valid IPv4 or IPv6 address.

```
>>> ipaddress.ip_address('192.168.0.1')
IPv4Address('192.168.0.1')
>>> ipaddress.ip_address('2001:db8::')
IPv6Address('2001:db8::')
```

ipaddress.**ip_network**(*address*, *strict=True*)

Return an `IPv4Network` or `IPv6Network` object depending on the IP address passed as argument. *address* is a string or integer representing the IP network. Either IPv4 or IPv6 networks may be supplied; integers less than `2**32` will be considered to be IPv4 by default. *strict* is passed to `IPv4Network` or `IPv6Network` constructor. A `ValueError` is raised if *address* does not represent a valid IPv4 or IPv6 address, or if the network has host bits set.

```
>>> ipaddress.ip_network('192.168.0.0/28')
IPv4Network('192.168.0.0/28')
```

ipaddress.**ip_interface**(*address*)

Return an `IPv4Interface` or `IPv6Interface` object depending on the IP address passed as argument. *address* is a string or integer representing the IP address. Either IPv4 or IPv6 addresses may be supplied; integers less than `2**32` will be considered to be IPv4 by default. A `ValueError` is raised if *address* does not represent a valid IPv4 or IPv6 address.

Un inconvénient de ces fonctions pratiques est que le besoin de gérer à la fois IPv4 et IPv6 signifie que les messages d'erreur contiennent peu d'information sur l'erreur précise puisqu'elles ne peuvent pas deviner quel format entre IPv4 et IPv6 est voulu. Un compte-rendu d'erreur plus détaillé peut être obtenu en appelant directement le constructeur de classe pour la version voulue.

22.23.2 Adresses IP

Objets adresse

Les objets `IPv4Address` et `IPv6Address` ont beaucoup d'attributs en commun. Certains attributs qui n'ont du sens que pour des adresses IPv6 sont aussi implémentés par les objets `IPv4Address` pour faciliter l'écriture de code qui gère les 2 versions IP correctement. Les objets d'adresse sont *hachables* pour qu'ils puissent être utilisés comme des clés dans des dictionnaires.

class ipaddress.**IPv4Address**(*address*)

Construit une adresse IPv4. Une exception `AddressValueError` est levée si *address* n'est pas une adresse IPv4 valide.

Une adresse IPv4 valide est composée de :

1. Une chaîne en notation décimale par points, composée de quatre entiers décimaux dans la plage inclusive 0—255, séparés par des points (p. ex. `192.168.0.1`). Chaque entier représente un octet dans l'adresse. Les zéros en tête ne sont pas tolérés pour éviter toute confusion avec la notation octale.
2. Un entier qui tient dans 32 bits.
3. Un entier tassé dans un objet `bytes` de taille 4 (gros-boutiste).

```
>>> ipaddress.IPv4Address('192.168.0.1')
IPv4Address('192.168.0.1')
>>> ipaddress.IPv4Address(3232235521)
IPv4Address('192.168.0.1')
>>> ipaddress.IPv4Address(b'\xC0\xA8\x00\x01')
IPv4Address('192.168.0.1')
```

Modifié dans la version 3.8 : Les zéros en tête sont tolérés, même dans les cas ambigus qui ont l'apparence de notation octal.

Modifié dans la version 3.9.5 : Les zéros en tête ne sont plus tolérés et seront traités comme une erreur. Les chaînes d'adresses IPv4 sont maintenant analysées aussi strictement que dans dans la fonction glibc `inet_pton()`.

version

Numéro de version approprié : `4` pour IPv4, `6` pour IPv6.

Modifié dans la version 3.14 : Made available on the class.

max_prefixlen

Nombre total de bits dans la représentation d'adresse de cette version : `32` pour IPv4, `128` pour IPv6.

Le préfixe définit le nombre de bits en tête dans une adresse qui sont comparés pour déterminer si une adresse fait partie d'un réseau.

Modifié dans la version 3.14 : Made available on the class.

compressed

exploded

Une chaîne en notation décimale par points. Les zéros en tête ne sont jamais inclus dans la représentation.

Comme IPv4 ne définit pas une notation abrégée pour les adresses avec des octets définis à zéro, ces deux attributs sont toujours identiques à `str(addr)` pour les adresses IPv4. Exposer ces attributs rend plus simple l'écriture de code d'affichage qui peut gérer à la fois les adresses IPv4 et IPv6.

packed

La représentation binaire de cette adresse — un objet *bytes* avec une taille appropriée (gros-boutiste). La taille est de 4 octets pour IPv4 et 16 octets pour IPv6.

reverse_pointer

Nom de l'enregistrement DNS PTR inverse pour l'adresse IP, p. ex :

```
>>> ipaddress.ip_address("127.0.0.1").reverse_pointer
'1.0.0.127.in-addr.arpa'
>>> ipaddress.ip_address("2001:db8::1").reverse_pointer
'1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.8.b.d.0.1.0.0.2.ip6.arpa'
```

C'est le nom utilisé pour effectuer une recherche inverse PTR et non le nom de l'hôte résolu lui-même.

Ajouté dans la version 3.5.

is_multicast

`True` si l'adresse est réservée pour utilisation *multicast*. Voir **RFC 3171** (pour IPv4) or **RFC 2373** (pour IPv6).

is_private

`True` if the address is defined as not globally reachable by iana-ipv4-special-registry (for IPv4) or iana-ipv6-special-registry (for IPv6) with the following exceptions :

- `is_private` is `False` for the shared address space (`100.64.0.0/10`)
- For IPv4-mapped IPv6-addresses the `is_private` value is determined by the semantics of the underlying IPv4 addresses and the following condition holds (see *IPv6Address.ipv4_mapped*) :

```
address.is_private == address.ipv4_mapped.is_private
```

`is_private` has value opposite to *is_global*, except for the shared address space (`100.64.0.0/10` range) where they are both `False`.

Modifié dans la version 3.13 : Fixed some false positives and false negatives.

- `192.0.0.0/24` is considered private with the exception of `192.0.0.9/32` and `192.0.0.10/32` (previously : only the `192.0.0.0/29` sub-range was considered private).
- `64:ff9b:1::/48` is considered private.
- `2002::/16` is considered private.
- There are exceptions within `2001::/23` (otherwise considered private) : `2001:1::1/128`, `2001:1::2/128`, `2001:3::/32`, `2001:4:112::/48`, `2001:20::/28`, `2001:30::/28`. The exceptions are not considered private.

is_global

`True` if the address is defined as globally reachable by iana-ipv4-special-registry (for IPv4) or iana-ipv6-special-registry (for IPv6) with the following exception :

For IPv4-mapped IPv6-addresses the `is_private` value is determined by the semantics of the underlying IPv4 addresses and the following condition holds (see *IPv6Address.ipv4_mapped*) :

```
address.is_global == address.ipv4_mapped.is_global
```

`is_global` has value opposite to *`is_private`*, except for the shared address space (`100.64.0.0/10` range) where they are both `False`.

Ajouté dans la version 3.4.

Modifié dans la version 3.13 : Fixed some false positives and false negatives, see *`is_private`* for details.

`is_unspecified`

`True` si l'adresse est non-spécifiée. Voir **RFC 5735** (pour IPv4) or **RFC 2373** (pour IPv6).

`is_reserved`

`True` si l'adresse est réservée par l'IETF.

`is_loopback`

`True` si c'est une adresse de rebouclage (*loopback* en anglais). Voir **RFC 3330** (pour IPv4) or **RFC 2373** (pour IPv6).

`is_link_local`

`True` si l'adresse est réservée pour les liaisons locales auto-configurées. Voir **RFC 3927**.

`ipv6_mapped`

`IPv4Address` object representing the IPv4-mapped IPv6 address. See **RFC 4291**.

Ajouté dans la version 3.13.

`IPv4Address.`**`__format__`**(*fmt*)

Renvoie l'adresse IP sous forme d'une chaîne, contrôlée par une chaîne de formatage explicite. *fmt* peut être : `'s'`, l'option par défaut, équivalent à *`str()`*, `'b'` pour une chaîne binaire préfixée par des zéros, `'X'` ou `'x'` pour une représentation hexadécimale majuscule ou minuscule, ou `'n'`, équivalant à `'b'` pour les adresses IPv4 et `'x'` pour IPv6. Pour les représentations binaires et hexadécimales, le spécificateur de forme `'#'` et l'option de regroupement `'_'` sont disponibles. `__format__` est utilisé par `format`, `str.format` et les f-strings.

```
>>> format(ipaddress.IPv4Address('192.168.0.1'))
'192.168.0.1'
>>> '{:#b}'.format(ipaddress.IPv4Address('192.168.0.1'))
'0b11000000101010000000000000000001'
>>> f'{ipaddress.IPv6Address("2001:db8::1000"):s}'
'2001:db8::1000'
>>> format(ipaddress.IPv6Address('2001:db8::1000'), '_X')
'2001_0DB8_0000_0000_0000_0000_0000_1000'
>>> '{:#_n}'.format(ipaddress.IPv6Address('2001:db8::1000'))
'0x2001_0db8_0000_0000_0000_0000_0000_1000'
```

Ajouté dans la version 3.9.

class `ipaddress.`**`IPv6Address`**(*address*)

Construit une adresse IPv6. Une *`AddressValueError`* est levée si *address* n'est pas une adresse IPv6 valide.

Une adresse IPv6 valide est constituée de :

1. Une chaîne constituée de huit groupes de quatre chiffres hexadécimaux, chaque groupe représentant 16 bits. Les groupes sont séparés par des deux-points. Ceci décrit une notation *éclatée* (longue). La chaîne peut-être aussi *abrégée* (notation courte) par différents moyens. Voir **RFC 4291** pour plus de détails. Par exemple, `"0000:0000:0000:0000:0000:0abc:0007:0def"` peut s'écrire `"::abc:7:def"`.

   Optionnellement, la chaîne peut avoir un indice de zone de portée, exprimé avec un suffixe `%scope_id`. Si présent, l'indice de portée ne doit pas être vide et ne doit pas contenir `%`. Voir **RFC 4007** pour plus de détails. Par exemple, `fe80::1234%1` pourrait identifier `fe80::1234` sur la première interface du nœud.
2. Un entier qui tient dans 128 bits.
3. Un entier tassé dans un objet *`bytes`* de taille 16, gros-boutiste.

```
>>> ipaddress.IPv6Address('2001:db8::1000')
IPv6Address('2001:db8::1000')
>>> ipaddress.IPv6Address('ff02::5678%1')
IPv6Address('ff02::5678%1')
```

compressed

Version courte de la représentation d'adresse, avec les groupes de zéros en tête omis et la séquence la plus longue de groupes constitués entièrement de zéros réduit à un seul groupe vide.

C'est aussi la valeur renvoyée par `str(addr)` pour les adresses IPv6.

exploded

Version longue de la représentation d'adresse, avec tous les zéros en tête et groupes composés entièrement de zéros inclus.

Pour les attributs et méthodes suivants, voir la documentation de la classe *`IPv4Address`* :

packed

reverse_pointer

version

max_prefixlen

is_multicast

is_private

is_global

Ajouté dans la version 3.4.

is_unspecified

is_reserved

is_loopback

is_link_local

is_site_local

`True` si l'adresse est réservée pour usage sur réseau local. Notez que l'espace d'adressage sur réseau local a été rendu obsolète par **RFC 3879**. Utilisez *`is_private`* pour tester si l'adresse est dans l'espace d'adresses locales et uniques défini par **RFC 4193**.

ipv4_mapped

Pour les adresses qui semblent être des adresses mappées IPv4 (commençant par `::FFFF/96`), cette propriété rapporte l'adresse IPv4 imbriquée. Pour toute autre adresse, cette propriété sera `None`.

scope_id

Pour les adresses avec une portée spécifiée comme définies par **RFC 4007**, cette propriété identifie la zone précise de la portée d'adresse à laquelle cette adresse appartient, en tant que chaîne. Quand la zone de portée n'est pas spécifiée, cette propriété est `None`.

sixtofour

Pour les adresses qui semblent être des adresses *6to4* (commençant par `2002::/16`) tel que défini par **RFC 3056**, cette propriété rapporte l'adresse IPv4 imbriquée. Pour toute autre adresse, cette propriété sera `None`.

teredo

Pour les adresses qui semblent être des adresses *Teredo* (commençant par `2001::/32`) tel que défini par **RFC 4380**, cette propriété rapporte la paire `(server, client)` imbriquée. Pour toute autre adresse, cette propriété sera `None`.

`IPv6Address.__format__(fmt)`

Référez-vous à la documentation de la méthode correspondante dans `IPv4Address`.

Ajouté dans la version 3.9.

Conversion vers les chaînes et les entiers

Afin d'interagir avec les API de réseau tels que le module socket, les adresses doivent être converties en chaînes et en entiers. Ceci est géré en utilisant les fonctions natives `str()` et `int()` :

```
>>> str(ipaddress.IPv4Address('192.168.0.1'))
'192.168.0.1'
>>> int(ipaddress.IPv4Address('192.168.0.1'))
3232235521
>>> str(ipaddress.IPv6Address('::1'))
'::1'
>>> int(ipaddress.IPv6Address('::1'))
1
```

Note that IPv6 scoped addresses are converted to integers without scope zone ID.

Opérateurs

Address objects support some operators. Unless stated otherwise, operators can only be applied between compatible objects (i.e. IPv4 with IPv4, IPv6 with IPv6).

Opérateurs de comparaison

Address objects can be compared with the usual set of comparison operators. Same IPv6 addresses with different scope zone IDs are not equal. Some examples :

```
>>> IPv4Address('127.0.0.2') > IPv4Address('127.0.0.1')
True
>>> IPv4Address('127.0.0.2') == IPv4Address('127.0.0.1')
False
>>> IPv4Address('127.0.0.2') != IPv4Address('127.0.0.1')
True
>>> IPv6Address('fe80::1234') == IPv6Address('fe80::1234%1')
False
>>> IPv6Address('fe80::1234%1') != IPv6Address('fe80::1234%2')
True
```

Opérateurs arithmétiques

Les entiers peuvent être additionnés ou soustraits des objets d'adresse. Quelques exemples :

```
>>> IPv4Address('127.0.0.2') + 3
IPv4Address('127.0.0.5')
>>> IPv4Address('127.0.0.2') - 3
IPv4Address('126.255.255.255')
>>> IPv4Address('255.255.255.255') + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ipaddress.AddressValueError: 4294967296 (>= 2**32) is not permitted as an IPv4␣
↪address
```

22.23.3 Définitions de réseaux IP

The `IPv4Network` and `IPv6Network` objects provide a mechanism for defining and inspecting IP network definitions. A network definition consists of a *mask* and a *network address*, and as such defines a range of IP addresses that equal the network address when masked (binary AND) with the mask. For example, a network definition with the mask `255.255.255.0` and the network address `192.168.1.0` consists of IP addresses in the inclusive range `192.168.1.0` to `192.168.1.255`.

Préfixe, masque réseau et masque de l'hôte

There are several equivalent ways to specify IP network masks. A *prefix* `/<nbits>` is a notation that denotes how many high-order bits are set in the network mask. A *net mask* is an IP address with some number of high-order bits set. Thus the prefix `/24` is equivalent to the net mask `255.255.255.0` in IPv4, or `ffff:ff00::` in IPv6. In addition, a *host mask* is the logical inverse of a *net mask*, and is sometimes used (for example in Cisco access control lists) to denote a network mask. The host mask equivalent to `/24` in IPv4 is `0.0.0.255`.

Objets réseau

All attributes implemented by address objects are implemented by network objects as well. In addition, network objects implement additional attributes. All of these are common between `IPv4Network` and `IPv6Network`, so to avoid duplication they are only documented for `IPv4Network`. Network objects are *hashable*, so they can be used as keys in dictionaries.

class ipaddress.**IPv4Network**(*address*, *strict=True*)

Construit une définition de réseau IPv4. *address* peut valoir :

1. A string consisting of an IP address and an optional mask, separated by a slash (`/`). The IP address is the network address, and the mask can be either a single number, which means it's a *prefix*, or a string representation of an IPv4 address. If it's the latter, the mask is interpreted as a *net mask* if it starts with a non-zero field, or as a *host mask* if it starts with a zero field, with the single exception of an all-zero mask which is treated as a *net mask*. If no mask is provided, it's considered to be `/32`.

   For example, the following *address* specifications are equivalent : `192.168.1.0/24`, `192.168.1.0/255.255.255.0` and `192.168.1.0/0.0.0.255`.
2. An integer that fits into 32 bits. This is equivalent to a single-address network, with the network address being *address* and the mask being `/32`.
3. An integer packed into a `bytes` object of length 4, big-endian. The interpretation is similar to an integer *address*.
4. A two-tuple of an address description and a netmask, where the address description is either a string, a 32-bits integer, a 4-bytes packed integer, or an existing `IPv4Address` object ; and the netmask is either an integer representing the prefix length (e.g. `24`) or a string representing the prefix mask (e.g. `255.255.255.0`).

An `AddressValueError` is raised if *address* is not a valid IPv4 address. A `NetmaskValueError` is raised if the mask is not valid for an IPv4 address.

If *strict* is `True` and host bits are set in the supplied address, then `ValueError` is raised. Otherwise, the host bits are masked out to determine the appropriate network address.

Unless stated otherwise, all network methods accepting other network/address objects will raise `TypeError` if the argument's IP version is incompatible to `self`.

Modifié dans la version 3.5 : Ajout de la forme paire pour le paramètre *address* du constructeur.

version

max_prefixlen

Référez-vous à la documentation de l'attribut correspondant dans `IPv4Address`.

is_multicast

is_private

is_unspecified

is_reserved

is_loopback

is_link_local

These attributes are true for the network as a whole if they are true for both the network address and the broadcast address.

network_address

The network address for the network. The network address and the prefix length together uniquely define a network.

broadcast_address

The broadcast address for the network. Packets sent to the broadcast address should be received by every host on the network.

hostmask

Le masque de l'hôte, en tant qu'objet *IPv4Address*.

netmask

Le masque réseau, en tant qu'objet *IPv4Address*.

with_prefixlen

compressed

exploded

Adresse IP du réseau sous forme d'une chaîne, avec le masque en notation CIDR.

`with_prefixlen` and `compressed` are always the same as `str(network)`. `exploded` uses the exploded form the network address.

with_netmask

A string representation of the network, with the mask in net mask notation.

with_hostmask

A string representation of the network, with the mask in host mask notation.

num_addresses

Le nombre total d'adresses dans le réseau.

prefixlen

Length of the network prefix, in bits.

hosts()

Returns an iterator over the usable hosts in the network. The usable hosts are all the IP addresses that belong to the network, except the network address itself and the network broadcast address. For networks with a mask length of 31, the network address and network broadcast address are also included in the result. Networks with a mask of 32 will return a list containing the single host address.

```
>>> list(ip_network('192.0.2.0/29').hosts())
[IPv4Address('192.0.2.1'), IPv4Address('192.0.2.2'),
 IPv4Address('192.0.2.3'), IPv4Address('192.0.2.4'),
 IPv4Address('192.0.2.5'), IPv4Address('192.0.2.6')]
>>> list(ip_network('192.0.2.0/31').hosts())
[IPv4Address('192.0.2.0'), IPv4Address('192.0.2.1')]
>>> list(ip_network('192.0.2.1/32').hosts())
[IPv4Address('192.0.2.1')]
```

overlaps(*other*)

`True` si ce réseau est partiellement ou complètement contenu dans *other* ou *other* est complètement contenu dans ce réseau.

address_exclude(*network*)

Computes the network definitions resulting from removing the given *network* from this one. Returns an iterator of network objects. Raises `ValueError` if *network* is not completely contained in this network.

```
>>> n1 = ip_network('192.0.2.0/28')
>>> n2 = ip_network('192.0.2.1/32')
>>> list(n1.address_exclude(n2))
[IPv4Network('192.0.2.8/29'), IPv4Network('192.0.2.4/30'),
 IPv4Network('192.0.2.2/31'), IPv4Network('192.0.2.0/32')]
```

subnets(*prefixlen_diff=1*, *new_prefix=None*)

The subnets that join to make the current network definition, depending on the argument values. *prefixlen_diff* is the amount our prefix length should be increased by. *new_prefix* is the desired new prefix of the subnets ; it must be larger than our prefix. One and only one of *prefixlen_diff* and *new_prefix* must be set. Returns an iterator of network objects.

```
>>> list(ip_network('192.0.2.0/24').subnets())
[IPv4Network('192.0.2.0/25'), IPv4Network('192.0.2.128/25')]
>>> list(ip_network('192.0.2.0/24').subnets(prefixlen_diff=2))
[IPv4Network('192.0.2.0/26'), IPv4Network('192.0.2.64/26'),
 IPv4Network('192.0.2.128/26'), IPv4Network('192.0.2.192/26')]
>>> list(ip_network('192.0.2.0/24').subnets(new_prefix=26))
[IPv4Network('192.0.2.0/26'), IPv4Network('192.0.2.64/26'),
 IPv4Network('192.0.2.128/26'), IPv4Network('192.0.2.192/26')]
>>> list(ip_network('192.0.2.0/24').subnets(new_prefix=23))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    raise ValueError('new prefix must be longer')
ValueError: new prefix must be longer
>>> list(ip_network('192.0.2.0/24').subnets(new_prefix=25))
[IPv4Network('192.0.2.0/25'), IPv4Network('192.0.2.128/25')]
```

supernet(*prefixlen_diff=1*, *new_prefix=None*)

The supernet containing this network definition, depending on the argument values. *prefixlen_diff* is the amount our prefix length should be decreased by. *new_prefix* is the desired new prefix of the supernet ; it must be smaller than our prefix. One and only one of *prefixlen_diff* and *new_prefix* must be set. Returns a single network object.

```
>>> ip_network('192.0.2.0/24').supernet()
IPv4Network('192.0.2.0/23')
>>> ip_network('192.0.2.0/24').supernet(prefixlen_diff=2)
IPv4Network('192.0.0.0/22')
>>> ip_network('192.0.2.0/24').supernet(new_prefix=20)
IPv4Network('192.0.0.0/20')
```

subnet_of(*other*)

Renvoie `True` si ce réseau est un sous-réseau de *other*.

```
>>> a = ip_network('192.168.1.0/24')
>>> b = ip_network('192.168.1.128/30')
>>> b.subnet_of(a)
True
```

Ajouté dans la version 3.7.

supernet_of(*other*)

Renvoie `True` si *other* est un sous-réseau de ce réseau.

```
>>> a = ip_network('192.168.1.0/24')
>>> b = ip_network('192.168.1.128/30')
>>> a.supernet_of(b)
True
```

Ajouté dans la version 3.7.

compare_networks(*other*)

Compare this network to *other*. In this comparison only the network addresses are considered ; host bits aren't. Returns either `-1`, `0` or `1`.

```
>>> ip_network('192.0.2.1/32').compare_networks(ip_network('192.0.2.2/32'))
-1
>>> ip_network('192.0.2.1/32').compare_networks(ip_network('192.0.2.0/32'))
1
>>> ip_network('192.0.2.1/32').compare_networks(ip_network('192.0.2.1/32'))
0
```

Obsolète depuis la version 3.7 : Utilise le même algorithme de relation d'ordre et de comparaison que <, ==, et >.

class ipaddress.**IPv6Network**(*address*, *strict=True*)

Construit une définition de réseau IPv6. *address* peut valoir :

1. A string consisting of an IP address and an optional prefix length, separated by a slash (`/`). The IP address is the network address, and the prefix length must be a single number, the *prefix*. If no prefix length is provided, it's considered to be `/128`.

   Note that currently expanded netmasks are not supported. That means `2001:db00::0/24` is a valid argument while `2001:db00::0/ffff:ff00::` is not.
2. An integer that fits into 128 bits. This is equivalent to a single-address network, with the network address being *address* and the mask being `/128`.
3. An integer packed into a `bytes` object of length 16, big-endian. The interpretation is similar to an integer *address*.
4. A two-tuple of an address description and a netmask, where the address description is either a string, a 128-bits integer, a 16-bytes packed integer, or an existing `IPv6Address` object ; and the netmask is an integer representing the prefix length.

An `AddressValueError` is raised if *address* is not a valid IPv6 address. A `NetmaskValueError` is raised if the mask is not valid for an IPv6 address.

If *strict* is `True` and host bits are set in the supplied address, then `ValueError` is raised. Otherwise, the host bits are masked out to determine the appropriate network address.

Modifié dans la version 3.5 : Ajout de la forme paire pour le paramètre *address* du constructeur.

version

max_prefixlen

is_multicast

is_private

is_unspecified

is_reserved

is_loopback

is_link_local

network_address

broadcast_address

hostmask

netmask

with_prefixlen

compressed

exploded

with_netmask

with_hostmask

num_addresses

prefixlen

hosts()

Returns an iterator over the usable hosts in the network. The usable hosts are all the IP addresses that belong to the network, except the Subnet-Router anycast address. For networks with a mask length of 127, the Subnet-Router anycast address is also included in the result. Networks with a mask of 128 will return a list containing the single host address.

overlaps(*other*)

address_exclude(*network*)

subnets(*prefixlen_diff=1*, *new_prefix=None*)

supernet(*prefixlen_diff=1*, *new_prefix=None*)

subnet_of(*other*)

supernet_of(*other*)

compare_networks(*other*)

Référez-vous à la documentation de l'attribut correspondant dans `IPv4Network`.

is_site_local

This attribute is true for the network as a whole if it is true for both the network address and the broadcast address.

Opérateurs

Network objects support some operators. Unless stated otherwise, operators can only be applied between compatible objects (i.e. IPv4 with IPv4, IPv6 with IPv6).

Opérateurs logiques

Network objects can be compared with the usual set of logical operators. Network objects are ordered first by network address, then by net mask.

Itération

Network objects can be iterated to list all the addresses belonging to the network. For iteration, *all* hosts are returned, including unusable hosts (for usable hosts, use the `hosts()` method). An example :

```
>>> for addr in IPv4Network('192.0.2.0/28'):
...     addr
...
IPv4Address('192.0.2.0')
IPv4Address('192.0.2.1')
IPv4Address('192.0.2.2')
IPv4Address('192.0.2.3')
IPv4Address('192.0.2.4')
IPv4Address('192.0.2.5')
IPv4Address('192.0.2.6')
IPv4Address('192.0.2.7')
IPv4Address('192.0.2.8')
IPv4Address('192.0.2.9')
IPv4Address('192.0.2.10')
IPv4Address('192.0.2.11')
IPv4Address('192.0.2.12')
IPv4Address('192.0.2.13')
IPv4Address('192.0.2.14')
IPv4Address('192.0.2.15')
```

Réseaux en tant que conteneurs d'adresses

Les objets réseau peuvent agir en tant que conteneurs d'adresses. Quelques exemples :

```
>>> IPv4Network('192.0.2.0/28')[0]
IPv4Address('192.0.2.0')
>>> IPv4Network('192.0.2.0/28')[15]
IPv4Address('192.0.2.15')
>>> IPv4Address('192.0.2.6') in IPv4Network('192.0.2.0/28')
True
>>> IPv4Address('192.0.3.6') in IPv4Network('192.0.2.0/28')
False
```

22.23.4 Objets interface

Les objets interface sont *hachables*, ce qui signifie qu'ils peuvent être utilisés comme clés de dictionnaire.

class ipaddress.**IPv4Interface**(*address*)

Construct an IPv4 interface. The meaning of *address* is as in the constructor of `IPv4Network`, except that arbitrary host addresses are always accepted.

`IPv4Interface` is a subclass of `IPv4Address`, so it inherits all the attributes from that class. In addition, the following attributes are available :

ip

L'adresse (`IPv4Address`) sans information de réseau.

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.ip
IPv4Address('192.0.2.5')
```

network

Le réseau (`IPv4Network`) auquel cette interface appartient.

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.network
IPv4Network('192.0.2.0/24')
```

with_prefixlen

A string representation of the interface with the mask in prefix notation.

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.with_prefixlen
'192.0.2.5/24'
```

with_netmask

A string representation of the interface with the network as a net mask.

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.with_netmask
'192.0.2.5/255.255.255.0'
```

with_hostmask

A string representation of the interface with the network as a host mask.

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.with_hostmask
'192.0.2.5/0.0.0.255'
```

class ipaddress.**IPv6Interface**(*address*)

Construct an IPv6 interface. The meaning of *address* is as in the constructor of `IPv6Network`, except that arbitrary host addresses are always accepted.

`IPv6Interface` is a subclass of `IPv6Address`, so it inherits all the attributes from that class. In addition, the following attributes are available :

ip

network

with_prefixlen

with_netmask

with_hostmask

Référez-vous à la documentation de l'attribut correspondant dans `IPv4Interface`.

Opérateurs

Interface objects support some operators. Unless stated otherwise, operators can only be applied between compatible objects (i.e. IPv4 with IPv4, IPv6 with IPv6).

Opérateurs logiques

Interface objects can be compared with the usual set of logical operators.

For equality comparison (== and !=), both the IP address and network must be the same for the objects to be equal. An interface will not compare equal to any address or network object.

For ordering (<, >, etc) the rules are different. Interface and address objects with the same IP version can be compared, and the address objects will always sort before the interface objects. Two interface objects are first compared by their networks and, if those are the same, then by their IP addresses.

22.23.5 Autres fonctions au niveau de module

Le module fournit aussi les fonctions suivantes :

ipaddress.**v4_int_to_packed**(*address*)

Represent an address as 4 packed bytes in network (big-endian) order. *address* is an integer representation of an IPv4 IP address. A `ValueError` is raised if the integer is negative or too large to be an IPv4 IP address.

```
>>> ipaddress.ip_address(3221225985)
IPv4Address('192.0.2.1')
>>> ipaddress.v4_int_to_packed(3221225985)
b'\xc0\x00\x02\x01'
```

ipaddress.**v6_int_to_packed**(*address*)

Represent an address as 16 packed bytes in network (big-endian) order. *address* is an integer representation of an IPv6 IP address. A `ValueError` is raised if the integer is negative or too large to be an IPv6 IP address.

ipaddress.**summarize_address_range**(*first*, *last*)

Return an iterator of the summarized network range given the first and last IP addresses. *first* is the first `IPv4Address` or `IPv6Address` in the range and *last* is the last `IPv4Address` or `IPv6Address` in the range. A `TypeError` is raised if *first* or *last* are not IP addresses or are not of the same version. A `ValueError` is raised if *last* is not greater than *first* or if *first* address version is not 4 or 6.

```
>>> [ipaddr for ipaddr in ipaddress.summarize_address_range(
...     ipaddress.IPv4Address('192.0.2.0'),
...     ipaddress.IPv4Address('192.0.2.130'))]
[IPv4Network('192.0.2.0/25'), IPv4Network('192.0.2.128/31'), IPv4Network('192.
↪0.2.130/32')]
```

ipaddress.**collapse_addresses**(*addresses*)

Return an iterator of the collapsed `IPv4Network` or `IPv6Network` objects. *addresses* is an *iterable* of `IPv4Network` or `IPv6Network` objects. A `TypeError` is raised if *addresses* contains mixed version objects.

```
>>> [ipaddr for ipaddr in
... ipaddress.collapse_addresses([ipaddress.IPv4Network('192.0.2.0/25'),
... ipaddress.IPv4Network('192.0.2.128/25')])]
[IPv4Network('192.0.2.0/24')]
```

ipaddress.**get_mixed_type_key**(*obj*)

Return a key suitable for sorting between networks and addresses. Address and Network objects are not sortable by default ; they're fundamentally different, so the expression :

```
IPv4Address('192.0.2.0') <= IPv4Network('192.0.2.0/24')
```

doesn't make sense. There are some times however, where you may wish to have `ipaddress` sort these anyway. If you need to do this, you can use this function as the *key* argument to `sorted()`.

obj est un objet réseau ou adresse.

22.23.6 Exceptions personnalisées

To support more specific error reporting from class constructors, the module defines the following exceptions :

exception ipaddress.**AddressValueError**(*ValueError*)

Toute erreur de valeur liée à l'adresse.

exception ipaddress.**NetmaskValueError**(*ValueError*)

Toute erreur de valeur liée au masque réseau.

CHAPITRE 23

Services multimédia

Les modules documentés dans ce chapitre implémentent divers algorithmes ou interfaces principalement utiles pour les applications multimédia. Ils peuvent ne pas être disponibles sur votre installation. En voici un aperçu :

23.1 `wave` --- Read and write WAV files

Code source : Lib/wave.py

The `wave` module provides a convenient interface to the Waveform Audio "WAVE" (or "WAV") file format. Only uncompressed PCM encoded wave files are supported.

Modifié dans la version 3.12 : Support for `WAVE_FORMAT_EXTENSIBLE` headers was added, provided that the extended format is `KSDATAFORMAT_SUBTYPE_PCM`.

Le module `wave` définit la fonction et l'exception suivante :

`wave.open`(*file*, *mode=None*)

Si *file* est une chaîne de caractères, ouvre le fichier sous ce nom, sinon, il est traité comme un objet de type fichier. *mode* peut être :

'rb'
Mode lecture seule.

'wb'
Mode écriture seule.

Notez que ce module ne permet pas de manipuler des fichiers WAV en lecture/écriture.

Un *mode* `'rb'` renvoie un objet `Wave_read`, alors qu'un *mode* `'wb'` renvoie un objet `Wave_write`. Si *mode* est omis et qu'un objet de type fichier est donné au paramètre *file*, `file.mode` est utilisé comme valeur par défaut pour *mode*.

If you pass in a file-like object, the wave object will not close it when its `close()` method is called ; it is the caller's responsibility to close the file object.

The `open()` function may be used in a `with` statement. When the `with` block completes, the `Wave_read.close()` or `Wave_write.close()` method is called.

Modifié dans la version 3.4 : Ajout de la gestion des fichiers non navigables.

exception wave.**Error**

Une erreur est levée lorsque quelque chose est impossible car elle enfreint la spécification WAV ou rencontre un problème d'implémentation.

23.1.1 Objets Wave_read

class wave.**Wave_read**

Read a WAV file.

Les objets Wave_read, tels qu'ils sont renvoyés par *open()*, ont les méthodes suivantes :

close()

Ferme le flux s'il a été ouvert par *wave* et rend l'instance inutilisable. Ceci est appelé automatiquement lorsque l'objet est détruit .

getnchannels()

Renvoie le nombre de canaux audio (`1` pour mono, `2` pour stéréo).

getsampwidth()

Renvoie la largeur de l'échantillon en octets.

getframerate()

Renvoie la fréquence d'échantillonnage.

getnframes()

Renvoie le nombre de trames audio.

getcomptype()

Renvoie le type de compression (`'NONE'` est le seul type géré).

getcompname()

Version compréhensible de *getcomptype()*. Généralement, `'not compressed'` équivaut à `'NONE'`.

getparams()

Returns a *namedtuple()* `(nchannels, sampwidth, framerate, nframes, comptype, compname)`, equivalent to output of the `get*()` methods.

readframes(*n*)

Lit et renvoie au plus *n* trames audio, sous forme d'objet *bytes*.

rewind()

Remet le pointeur de fichier au début du flux audio.

The following two methods are defined for compatibility with the old `aifc` module, and don't do anything interesting.

getmarkers()

Renvoie `None`.

Deprecated since version 3.13, will be removed in version 3.15 : The method only existed for compatibility with the `aifc` module which has been removed in Python 3.13.

getmark(*id*)

Lève une erreur.

Deprecated since version 3.13, will be removed in version 3.15 : The method only existed for compatibility with the `aifc` module which has been removed in Python 3.13.

Les deux fonctions suivantes utilisent le vocabulaire "position". Ces positions sont compatible entre elles, la "position" de l'un est compatible avec la "position" de l'autre. Cette position est dépendante de l'implémentation.

setpos(*pos*)

Place le pointeur du fichier sur la position spécifiée.

tell()

Renvoie la position actuelle du pointeur du fichier.

23.1.2 Objets Wave_write

class wave.**Wave_write**

Write a WAV file.

Wave_write objects, as returned by `open()`.

For seekable output streams, the `wave` header will automatically be updated to reflect the number of frames actually written. For unseekable streams, the *nframes* value must be accurate when the first frame data is written. An accurate *nframes* value can be achieved either by calling `setnframes()` or `setparams()` with the number of frames that will be written before `close()` is called and then using `writeframesraw()` to write the frame data, or by calling `writeframes()` with all of the frame data to be written. In the latter case `writeframes()` will calculate the number of frames in the data and set *nframes* accordingly before writing the frame data.

Modifié dans la version 3.4 : Ajout de la gestion des fichiers non navigables.

Wave_write objects have the following methods :

close()

Assurez-vous que *nframes* soit correct et fermez le fichier s'il a été ouvert par `wave`. Cette méthode est appelée à la destruction de l'objet. Il lève une erreur si le flux de sortie n'est pas navigable et si *nframes* ne correspond pas au nombre de trames réellement écrites.

setnchannels(*n*)

Définit le nombre de canaux.

setsampwidth(*n*)

Définit la largeur de l'échantillon à *n* octets.

setframerate(*n*)

Définit la fréquence des trames à *n*.

Modifié dans la version 3.2 : Un paramètre non-entier passé à cette méthode est arrondi à l'entier le plus proche.

setnframes(*n*)

Définit le nombre de trames à *n*. Cela sera modifié ultérieurement si le nombre de trames réellement écrites est différent (la tentative de mise à jour générera une erreur si le flux de sortie n'est pas indexable).

setcomptype(*type*, *name*)

Définit le type de compression et la description. Pour le moment, seul le type de compression `NONE` est géré, c'est-à-dire aucune compression.

setparams(*tuple*)

The *tuple* should be `(nchannels, sampwidth, framerate, nframes, comptype, compname)`, with values valid for the `set*()` methods. Sets all parameters.

tell()

Renvoie la position actuelle dans le fichier, avec les mêmes réserves que pour les méthodes `Wave_read.tell()` et `Wave_read.setpos()`.

writeframesraw(*data*)

Écrit les trames audio sans corriger *nframes*.

Modifié dans la version 3.4 : N'importe quel *bytes-like object* est maintenant accepté.

writeframes(*data*)

Écrit des trames audio et s'assure que *nframes* est correct. Une erreur est levée si le flux de sortie est non-navigable et si le nombre total de trames écrites après que *data* soit écrit ne correspond pas à la valeur précédemment définie pour *nframes*.

Modifié dans la version 3.4 : N'importe quel *bytes-like object* est maintenant accepté.

Notez qu'il est impossible de définir des paramètres après avoir appelé `writeframes()` ou `writeframesraw()`, et toute tentative en ce sens lève une `wave.Error`.

23.2 `colorsys` --- Conversions between color systems

Code source : Lib/colorsys.py

The `colorsys` module defines bidirectional conversions of color values between colors expressed in the RGB (Red Green Blue) color space used in computer monitors and three other coordinate systems : YIQ, HLS (Hue Lightness Saturation) and HSV (Hue Saturation Value). Coordinates in all of these color spaces are floating-point values. In the YIQ space, the Y coordinate is between 0 and 1, but the I and Q coordinates can be positive or negative. In all other spaces, the coordinates are all between 0 and 1.

Voir aussi

Consultez https://poynton.ca/ColorFAQ.html et https://www.cambridgeincolour.com/tutorials/color-spaces.htm pour plus d'informations concernant les espaces colorimétriques.

Le module `colorsys` définit les fonctions suivantes :

`colorsys.rgb_to_yiq`(*r*, *g*, *b*)

Convertit la couleur des coordonnées RGB (RVB) vers les coordonnées YIQ.

`colorsys.yiq_to_rgb`(*y*, *i*, *q*)

Convertit la couleur des coordonnées YIQ vers les coordonnées RGB (RVB).

`colorsys.rgb_to_hls`(*r*, *g*, *b*)

Convertit la couleur des coordonnées RGB (RVB) vers les coordonnées HLS (TSV).

`colorsys.hls_to_rgb`(*h*, *l*, *s*)

Convertit la couleur des coordonnées HLS (TSV) vers les coordonnées RGB (RVB).

`colorsys.rgb_to_hsv`(*r*, *g*, *b*)

Convertit la couleur des coordonnées RGB (RVB) vers les coordonnées HSV (TSV).

`colorsys.hsv_to_rgb`(*h*, *s*, *v*)

Convertit la couleur des coordonnées HSV (TSV) vers les coordonnées RGB (RVB).

Exemple :

```
>>> import colorsys
>>> colorsys.rgb_to_hsv(0.2, 0.4, 0.4)
(0.5, 0.5, 0.4)
>>> colorsys.hsv_to_rgb(0.5, 0.5, 0.4)
(0.2, 0.4, 0.4)
```

CHAPITRE 24

Internationalisation

Les modules décrits dans ce chapitre vous aident à rédiger des programmes indépendants des langues et des cultures en fournissant des mécanismes pour sélectionner une langue à utiliser dans les messages, ou en adaptant la sortie aux conventions culturelles.

La liste des modules documentés dans ce chapitre est :

24.1 `gettext` --- Multilingual internationalization services

Code source : Lib/gettext.py

Le module `gettext` fournit un service d'internationalisation (*I18N*) et de localisation linguistique (*L10N*) pour vos modules et applications Python. Il est compatible avec l'API du catalogue de messages GNU **gettext** et à un plus haut niveau, avec l'API basée sur les classes qui serait peut-être plus adaptée aux fichiers Python. L'interface décrite ci-dessous vous permet d'écrire les textes de vos modules et applications dans une langue naturelle, puis de fournir un catalogue de traductions pour les lancer ensuite dans d'autres langues naturelles.

Quelques astuces sur la localisation de vos modules et applications Python sont également données.

24.1.1 API GNU `gettext`

Le module `gettext` définit l'API suivante, qui est très proche de l'API de GNU **gettext**. Si vous utilisez cette API, cela affectera la traduction de toute votre application. C'est souvent le comportement attendu si votre application est monolingue, avec le choix de la langue qui dépend des paramètres linguistiques de l'utilisateur. Si vous localisez un module Python ou si votre application a besoin de changer de langue à la volée, il est plus judicieux d'utiliser l'API basée sur des classes.

`gettext.`**`bindtextdomain`**(*domain*, *localedir=None*)

Lie *domain* au répertoire *localedir* des localisations. Plus spécifiquement, `gettext` va chercher les fichiers binaires `.mo` pour un domaine donné, en utilisant le chemin suivant (sous Unix) : *`localedir`*`/`*`language`*`/LC_MESSAGES/`*`domain`*`.mo`, où *language* est recherché dans l'une des variables d'environnement suivantes : `LANGUAGE`, `LC_ALL`, `LC_MESSAGES` et `LANG`.

Si *localedir* n'est pas renseigné ou vaut `None`, alors le lien actuel de *domain* est renvoyé. [1]

1. The default locale directory is system dependent ; for example, on Red Hat Linux it is `/usr/share/locale`, but on Solaris it is `/usr/lib/locale`. The `gettext` module does not try to support these system dependent defaults ; instead its default is *`sys.base_prefix`*`/share/locale` (see *`sys.base_prefix`*). For this reason, it is always best to call *`bindtextdomain()`* with an explicit absolute path at the start of your application.

gettext.**textdomain**(*domain=None*)

Change ou interroge le domaine global actuel. Si *domain* vaut `None`, alors le domaine global actuel est renvoyé. Sinon, le domaine global est positionné à *domain*, puis renvoyé.

gettext.**gettext**(*message*)

Return the localized translation of *message*, based on the current global domain, language, and locale directory. This function is usually aliased as `_()` in the local namespace (see examples below).

gettext.**dgettext**(*domain*, *message*)

Comme `gettext()`, mais cherche le message dans le domaine spécifié.

gettext.**ngettext**(*singular*, *plural*, *n*)

Comme `gettext()`, mais prend en compte les formes au pluriel. Si une traduction a été trouvée, utilise la formule pour trouver le pluriel à *n* et renvoie le message généré (quelques langues ont plus de deux formes au pluriel). Si aucune traduction n'a été trouvée, renvoie *singular* si *n* vaut 1, *plural* sinon.

La formule pour trouver le pluriel est récupérée dans l'entête du catalogue. C'est une expression en C ou en Python qui a une variable libre *n* et qui évalue l'index du pluriel dans le catalogue. Voir la documentation de GNU gettext pour la syntaxe précise à utiliser dans les fichiers `.po` et pour les formules dans différents langues.

gettext.**dngettext**(*domain*, *singular*, *plural*, *n*)

Comme `ngettext()`, mais cherche le message dans le domaine spécifié.

gettext.**pgettext**(*context*, *message*)

gettext.**dpgettext**(*domain*, *context*, *message*)

gettext.**npgettext**(*context*, *singular*, *plural*, *n*)

gettext.**dnpgettext**(*domain*, *context*, *singular*, *plural*, *n*)

Semblable aux fonctions correspondantes sans le `p` dans le préfixe (c'est-à-dire `gettext()`, `dgettext()`, `ngettext()` et `dngettext()`), mais la traduction est limitée au *context* du message donné.

Ajouté dans la version 3.8.

Note that GNU **gettext** also defines a `dcgettext()` method, but this was deemed not useful and so it is currently unimplemented.

Voici un exemple classique d'utilisation de cette API :

```
import gettext
gettext.bindtextdomain('myapplication', '/path/to/my/language/directory')
gettext.textdomain('myapplication')
_ = gettext.gettext
# ...
print(_('This is a translatable string.'))
```

24.1.2 API basée sur les classes

The class-based API of the `gettext` module gives you more flexibility and greater convenience than the GNU **gettext** API. It is the recommended way of localizing your Python applications and modules. `gettext` defines a `GNUTranslations` class which implements the parsing of GNU `.mo` format files, and has methods for returning strings. Instances of this class can also install themselves in the built-in namespace as the function `_()`.

gettext.**find**(*domain*, *localedir=None*, *languages=None*, *all=False*)

Cette fonction implémente l'algorithme standard de recherche de fichier `mo`. Il prend en entrée un *domain*, tout comme la fonction `textdomain()`. Le paramètre optionnel *localedir* est le même que celui de `bindtextdomain()`. Le paramètre optionnel *langages* est une liste de chaînes de caractères correspondants au code d'une langue.

Si *localedir* n'est pas renseigné, alors le répertoire de la locale par défaut du système est utilisé.[2] Si *languages* n'est pas renseigné, alors les variables d'environnement suivantes sont utilisées : `LANGUAGE`, `LC_ALL`,

2. Voir la note de `bindtextdomain()` ci-dessus.

LC_MESSAGES et LANG. La première à renvoyer une valeur non vide est alors utilisée pour *languages*. Ces variables d'environnement doivent contenir une liste de langues, séparées par des deux-points, qui sera utilisée pour générer la liste des codes de langues attendue.

Recherche avec `find()`, découvre et normalise les langues, puis itère sur la liste obtenue afin de trouver un fichier de traduction existant et correspondant :

localedir/*language*/LC_MESSAGES/*domain*.mo

Le premier nom de fichier trouvé est renvoyé par `find()`. Si aucun fichier n'a été trouvé, alors `None` est renvoyé. Si *all* est vrai, est renvoyée la liste de tous les noms de fichiers, dans l'ordre dans lequel ils apparaissent dans *languages* ou dans les variables d'environnement.

gettext.**translation**(*domain*, *localedir=None*, *languages=None*, *class_=None*, *fallback=False*)

Return a `*Translations` instance based on the *domain*, *localedir*, and *languages*, which are first passed to `find()` to get a list of the associated `.mo` file paths. Instances with identical `.mo` file names are cached. The actual class instantiated is *class_* if provided, otherwise `GNUTranslations`. The class's constructor must take a single *file object* argument.

Si plusieurs fichiers ont été trouvés, les derniers sont utilisés comme substitut des premiers. Pour rendre possible cette substitution, `copy.copy()` est utilisé pour copier chaque objet traduit depuis le cache ; les vraies données de l'instance étant toujours recopiées dans le cache.

Si aucun fichier `.mo` n'a été trouvé, soit *fallback* vaut *False* (valeur par défaut) et une exception `OSError` est levée, soit *fallback* vaut *True* et une instance `NullTranslations` est renvoyée.

Modifié dans la version 3.3 : `IOError` used to be raised, it is now an alias of `OSError`.

Modifié dans la version 3.11 : *codeset* parameter is removed.

gettext.**install**(*domain*, *localedir=None*, *, *names=None*)

This installs the function `_()` in Python's builtins namespace, based on *domain* and *localedir* which are passed to the function `translation()`.

Concernant le paramètre *names*, se référer à la description de la méthode `install()`.

As seen below, you usually mark the strings in your application that are candidates for translation, by wrapping them in a call to the `_()` function, like this :

```
print(_('This string will be translated.'))
```

For convenience, you want the `_()` function to be installed in Python's builtins namespace, so it is easily accessible in all modules of your application.

Modifié dans la version 3.11 : *names* est désormais un paramètre à mot-clé uniquement.

La classe `NullTranslations`

Les classes de traduction implémentent le fait de passer d'une chaîne de caractères du fichier original à traduire à la traduction de celle-ci. La classe de base utilisée est `NullTranslations`. C'est l'interface de base à utiliser lorsque vous souhaitez écrire vos propres classes spécifiques à la traduction. Voici les méthodes de `NullTranslations` :

class gettext.**NullTranslations**(*fp=None*)

Prend un paramètre optionnel un *file object* *fp*, qui est ignoré par la classe de base. Initialise les variables d'instance "protégées" *_info* et *_chardet*, définies par des classes dérivées, tout comme *_fallback* qui est définie au travers de `add_fallback()`. Puis appelle `self._parse(fp)` si *fp* ne vaut pas `None`.

_parse(*fp*)

Cette méthode, non exécutée dans la classe de base, prend en paramètre un objet fichier *fp* et lit les données de ce dernier. Si vous avez un catalogue de messages dont le format n'est pas pris en charge, vous devriez surcharger cette méthode pour analyser votre format.

add_fallback(*fallback*)

Ajoute *fallback* comme objet de substitution pour l'objet de traduction courant. Un objet de traduction devrait interroger cet objet de substitution s'il ne peut fournir une traduction pour un message donné.

gettext(*message*)

Si un objet de substitution a été défini, transmet `gettext()` à celui-ci. Sinon, renvoie *message*. Surchargé dans les classes dérivées.

ngettext(*singular*, *plural*, *n*)

Si un objet de substitution a été défini, transmet `ngettext()` à celui-ci. Sinon, renvoie *singular* si *n* vaut 1, *plural* sinon. Surchargé dans les classes dérivées.

pgettext(*context*, *message*)

Si un objet de substitution a été défini, transmet `pgettext()` à celui-ci. Sinon, renvoie le message traduit. Surchargé dans les classes dérivées.

Ajouté dans la version 3.8.

npgettext(*context*, *singular*, *plural*, *n*)

Si un objet de substitution a été défini, transmet `npgettext()` à celui-ci. Sinon, renvoie le message traduit. Surchargé dans les classes dérivées.

Ajouté dans la version 3.8.

info()

Return a dictionary containing the metadata found in the message catalog file.

charset()

Renvoie l'encodage du fichier du catalogue de messages.

install(*names=None*)

Cette méthode positionne `gettext()` dans l'espace de nommage natif, en le liant à `_`.

If the *names* parameter is given, it must be a sequence containing the names of functions you want to install in the builtins namespace in addition to `_()`. Supported names are `'gettext'`, `'ngettext'`, `'pgettext'`, and `'npgettext'`.

Note that this is only one way, albeit the most convenient way, to make the `_()` function available to your application. Because it affects the entire application globally, and specifically the built-in namespace, localized modules should never install `_()`. Instead, they should use this code to make `_()` available to their module :

```
import gettext
t = gettext.translation('mymodule', ...)
_ = t.gettext
```

This puts `_()` only in the module's global namespace and so only affects calls within this module.

Modifié dans la version 3.8 : Ajout de `'pgettext'` et `'npgettext'`.

La classe `GNUTranslations`

The `gettext` module provides one additional class derived from `NullTranslations` : `GNUTranslations`. This class overrides `_parse()` to enable reading GNU **gettext** format `.mo` files in both big-endian and little-endian format.

`GNUTranslations` parses optional metadata out of the translation catalog. It is convention with GNU **gettext** to include metadata as the translation for the empty string. This metadata is in **RFC 822**-style `key: value` pairs, and should contain the `Project-Id-Version` key. If the key `Content-Type` is found, then the `charset` property is used to initialize the "protected" `_charset` instance variable, defaulting to `None` if not found. If the charset encoding is specified, then all message ids and message strings read from the catalog are converted to Unicode using this encoding, else ASCII is assumed.

Since message ids are read as Unicode strings too, all `*gettext()` methods will assume message ids as Unicode strings, not byte strings.

The entire set of key/value pairs are placed into a dictionary and set as the "protected" `_info` instance variable.

Si le nombre magique du fichier `.mo` est invalide, le numéro de la version majeure inattendu, ou si d'autres problèmes apparaissent durant la lecture du fichier, instancier une classe `GNUTranslations` peut lever une exception `OSError`.

class gettext.**GNUTranslations**

Les méthodes suivantes, provenant de l'implémentation de la classe de base, ont été surchargée :

gettext(*message*)

Recherche l'identifiant de *message* dans le catalogue et renvoie le message de la chaîne de caractères correspondante comme une chaîne Unicode. Si aucun identifiant n'a été trouvé pour *message* et qu'un substitut a été défini, la recherche est transmise à la méthode `gettext()` du substitut. Sinon, l'identifiant de *message* est renvoyé.

ngettext(*singular*, *plural*, *n*)

Effectue une recherche sur les formes plurielles de l'identifiant d'un message. *singular* est utilisé pour la recherche de l'identifiant dans le catalogue, alors que *n* permet de savoir quelle forme plurielle utiliser. La chaîne de caractère du message renvoyée est une chaîne Unicode.

Si l'identifiant du message n'est pas trouvé dans le catalogue et qu'un substitut a été spécifié, la requête est transmise à la méthode `ngettext()` du substitut. Sinon, est renvoyé *singular* lorsque *n* vaut 1, *plural* dans tous les autres cas.

Voici un exemple :

```
n = len(os.listdir('.'))
cat = GNUTranslations(somefile)
message = cat.ngettext(
    'There is %(num)d file in this directory',
    'There are %(num)d files in this directory',
    n) % {'num': n}
```

pgettext(*context*, *message*)

Recherche le *contexte* et l'identifiant de *message* dans le catalogue et renvoie le message de la chaîne de caractères correspondante comme une chaîne Unicode. Si aucun identifiant n'a été trouvé pour l'identifiant du *message* et du *context* et qu'un substitut a été défini, la recherche est transmise à la méthode `pgettext()` du substitut. Sinon, l'identifiant de *message* est renvoyé.

Ajouté dans la version 3.8.

npgettext(*context*, *singular*, *plural*, *n*)

Effectue une recherche sur les formes plurielles de l'identifiant d'un message. *singular* est utilisé pour la recherche de l'identifiant dans le catalogue, alors que *n* permet de savoir quelle forme plurielle utiliser.

Si l'identifiant du message pour le *context* n'est pas trouvé dans le catalogue et qu'un substitut a été spécifié, la requête est transmise à la méthode `npgettext()` du substitut. Sinon, est renvoyé *singular* lorsque *n* vaut 1, *plural* dans tous les autres cas.

Ajouté dans la version 3.8.

Support du catalogue de message de Solaris

Le système d'exploitation Solaris possède son propre format de fichier binaire `.mo`, mais pour l'heure, puisqu'on ne peut trouver de documentation sur ce format, il n'est pas géré.

Le constructeur *Catalog*

GNOME utilise une version du module `gettext` de James Henstridge, mais qui a une API légèrement différente. D'après la documentation, elle s'utilise ainsi :

```
import gettext
cat = gettext.Catalog(domain, localedir)
```

(suite sur la page suivante)

(suite de la page précédente)

```
_ = cat.gettext
print(_('hello world'))
```

For compatibility with this older module, the function `Catalog()` is an alias for the *translation()* function described above.

Une différence entre ce module et celui de Henstridge : les objets de son catalogue étaient accessibles depuis un schéma de l'API, mais cela semblait ne pas être utilisé et donc n'est pas pris en charge.

24.1.3 Internationaliser vos programmes et modules

L'internationalisation (*I18N*) consiste à permettre à un programme de recevoir des traductions dans plusieurs langues. La localisation (*L10N*) consiste à adapter un programme à la langue et aux habitudes culturelles locales, une fois celui-ci internationalisé. Afin de fournir du texte multilingue à votre programme Python, les étapes suivantes sont nécessaires :

1. préparer votre programme ou module en marquant spécifiquement les chaînes à traduire
2. lancer une suite d'outils sur les fichiers contenant des chaînes à traduire pour générer des catalogues de messages brut
3. créer les traductions spécifiques à une langue des catalogues de messages
4. utiliser le module *gettext* pour que les chaînes de caractères soient bien traduites

In order to prepare your code for I18N, you need to look at all the strings in your files. Any string that needs to be translated should be marked by wrapping it in `_('...')` --- that is, a call to the function `_`. For example :

```
filename = 'mylog.txt'
message = _('writing a log message')
with open(filename, 'w') as fp:
    fp.write(message)
```

Dans cet exemple, la chaîne `'writing a log message'` est maquée comme traduite, contrairement aux chaînes `'mylog.txt'` et `'w'`.

There are a few tools to extract the strings meant for translation. The original GNU **gettext** only supported C or C++ source code but its extended version **xgettext** scans code written in a number of languages, including Python, to find strings marked as translatable. Babel is a Python internationalization library that includes a `pybabel` script to extract and compile message catalogs. François Pinard's program called **xpot** does a similar job and is available as part of his po-utils package.

(Python inclut également des versions en Python de ces programmes, **pygettext.py** et **msgfmt.py**, que certaines distributions Python installeront pour vous. **pygettext.py** est similaire à **xgettext**, mais ne comprend que le code source écrit en Python et ne peut prendre en charge d'autres langages de programmation tels que le C ou C++. **pygettext.py** possède une interface en ligne de commande similaire à celle de **xgettext** --- pour plus de détails sur son utilisation, exécuter `pygettext.py --help`. **msgfmt.py** est compatible avec GNU **msgfmt**. Avec ces deux programmes, vous ne devriez pas avoir besoin du paquet GNU **gettext** pour internationaliser vos applications en Python.)

xgettext, **pygettext** et d'autres outils similaires génèrent des fichiers `.po` représentant les catalogues de messages. Il s'agit de fichiers structurés et lisibles par un être humain, qui contiennent toutes les chaînes du code source marquées comme traduisible, ainsi que leur traduction à utiliser.

Les copies de ces fichiers `.po` sont ensuite remises à des êtres humains qui traduisent le contenu pour chaque langue naturelle prise en charge. Pour chacune des langues, ces derniers renvoient la version complétée sous la forme d'un fichier `<code-langue>.po` qui a été compilé dans un fichier binaire `.mo` représentant le catalogue lisible par une machine à l'aide du programme **msgfmt**. Les fichiers `.mo` sont utilisés par le module *gettext* pour la traduction lors de l'exécution.

La façon dont vous utilisez le module *gettext* dans votre code dépend de si vous internationalisez un seul module ou l'ensemble de votre application. Les deux sections suivantes traitent chacune des cas.

Localiser votre module

Si vous localisez votre module, veillez à ne pas faire de changements globaux, e.g. dans l'espace de nommage natif. Vous ne devriez pas utiliser l'API GNU **gettext** mais plutôt celle basée sur les classes.

Disons que votre module s'appelle "spam" et que les fichiers `.mo` de traduction dans les différentes langues naturelles soient dans `/usr/share/locale` au format GNU **gettext**. Voici ce que vous pouvez alors mettre en haut de votre module :

```
import gettext
t = gettext.translation('spam', '/usr/share/locale')
_ = t.gettext
```

Localiser votre application

If you are localizing your application, you can install the `_()` function globally into the built-in namespace, usually in the main driver file of your application. This will let all your application-specific files just use `_('...')` without having to explicitly install it in each file.

Dans ce cas, vous n'aurez à ajouter que le bout de code suivant au fichier principal de votre application :

```
import gettext
gettext.install('myapplication')
```

Si vous avez besoin de définir le dossier des localisations, vous pouvez le mettre en argument de la fonction *install()* :

```
import gettext
gettext.install('myapplication', '/usr/share/locale')
```

Changer de langue à la volée

Si votre programme a besoin de prendre en charge plusieurs langues en même temps, vous pouvez créer plusieurs instances de traduction, puis basculer entre elles de façon explicite, comme ceci :

```
import gettext

lang1 = gettext.translation('myapplication', languages=['en'])
lang2 = gettext.translation('myapplication', languages=['fr'])
lang3 = gettext.translation('myapplication', languages=['de'])

# start by using language1
lang1.install()

# ... time goes by, user selects language 2
lang2.install()

# ... more time goes by, user selects language 3
lang3.install()
```

Traductions différées

Dans la plupart des cas, en programmation, les chaînes de caractères sont traduites à l'endroit où on les écrit. Cependant, il peut arriver que vous ayez besoin de traduire une chaîne de caractères un peu plus loin. Un exemple classique est :

```
animals = ['mollusk',
           'albatross',
           'rat',
```

(suite sur la page suivante)

(suite de la page précédente)

```
           'penguin',
           'python', ]
# ...
for a in animals:
    print(a)
```

Ici, vous voulez marquer les chaînes de caractères de la liste `animals` comme étant traduisibles, mais ne les traduire qu'au moment de les afficher.

Voici un moyen de gérer ce cas :

```
def _(message): return message

animals = [_('mollusk'),
           _('albatross'),
           _('rat'),
           _('penguin'),
           _('python'), ]

del _

# ...
for a in animals:
    print(_(a))
```

This works because the dummy definition of `_()` simply returns the string unchanged. And this dummy definition will temporarily override any definition of `_()` in the built-in namespace (until the `del` command). Take care, though if you have a previous definition of `_()` in the local namespace.

Note that the second use of `_()` will not identify "a" as being translatable to the **gettext** program, because the parameter is not a string literal.

Voici une autre solution :

```
def N_(message): return message

animals = [N_('mollusk'),
           N_('albatross'),
           N_('rat'),
           N_('penguin'),
           N_('python'), ]

# ...
for a in animals:
    print(_(a))
```

In this case, you are marking translatable strings with the function `N_()`, which won't conflict with any definition of `_()`. However, you will need to teach your message extraction program to look for translatable strings marked with `N_()`. **xgettext**, **pygettext**, `pybabel extract`, and **xpot** all support this through the use of the `-k` command-line switch. The choice of `N_()` here is totally arbitrary; it could have just as easily been `MarkThisStringForTranslation()`.

24.1.4 Remerciements

Les personnes suivantes ont contribué au code, ont fait des retours, ont participé aux suggestions de conception et aux implémentations précédentes, et ont partagé leur expérience précieuse pour la création de ce module :

- Peter Funk
- James Henstridge
- Juan David Ibáñez Palomar

— Marc-André Lemburg
— Martin von Löwis
— François Pinard
— Barry Warsaw
— Gustavo Niemeyer

Notes

24.2 `locale` --- Internationalization services

Source code : Lib/locale.py

Le module `locale` donne accès à la base de données et aux fonctionnalités des paramètres linguistiques définis par POSIX. Le mécanisme des paramètres linguistiques de POSIX permet aux développeurs de faire face à certaines problématiques culturelles dans une application, sans avoir à connaître toutes les spécificités de chaque pays où le logiciel est exécuté.

The `locale` module is implemented on top of the `_locale` module, which in turn uses an ANSI C locale implementation if available.

Le module `locale` définit l'exception et les fonctions suivantes :

exception `locale.`**`Error`**

Exception levée lorsque le paramètre régional passé en paramètre de `setlocale()` n'est pas reconnu.

`locale.`**`setlocale`**(*category*, *locale=None*)

Si *locale* ne vaut pas `None`, `setlocale()` modifie le paramètre régional pour la catégorie *category*. Les catégories disponibles sont listées dans la description des données ci-dessous. *locale* peut être une chaîne de caractères ou un itérable de deux chaînes de caractères (code de la langue et encodage). Si c'est un itérable, il est converti en un nom de paramètre régional à l'aide du moteur de normalisation fait pour. Si c'est une chaîne vide, les paramètres par défaut de l'utilisateur sont utilisés. Si la modification du paramètre régional échoue, l'exception `Error` est levée. Si elle fonctionne, le nouveau paramètre est renvoyé.

Si *locale* est omis ou vaut `None`, le paramètre actuel de *category* est renvoyé.

`setlocale()` n'est pas *thread-safe* sur la plupart des systèmes. Les applications commencent généralement par un appel de

```
import locale
locale.setlocale(locale.LC_ALL, '')
```

Cela définit les paramètres régionaux dans toutes les catégories sur ceux par défaut de l'utilisateur (habituellement spécifiés dans la variable d'environnement `LANG`). Si les paramètres régionaux ne sont pas modifiés par la suite, l'utilisation de fils d'exécution ne devrait pas poser de problèmes.

`locale.`**`localeconv`**()

Renvoie la base de données des conventions locales sous forme de dictionnaire. Ce dictionnaire a les chaînes de caractères suivantes comme clés :

Catégorie	Clé	Signification
LC_NUMERIC	'decimal_point'	Caractère du séparateur décimal (entre la partie entière et la partie décimale).
	'grouping'	Séquence de nombres spécifiant les positions relatives attendues pour 'thousands_sep' (séparateur de milliers). Si la séquence se termine par *CHAR_MAX*, aucun autre regroupement n'est effectué. Si la séquence se termine par un 0, la dernière taille du groupe est utilisée à plusieurs reprises.
	'thousands_sep'	Caractère utilisé entre les groupes (séparateur de milliers).
LC_MONETARY	'int_curr_symbol'	Symbole monétaire international.
	'currency_symbol'	Symbole monétaire local.
	'p_cs_precedes/n_cs_precedes'	Si le symbole monétaire précède ou non la valeur (pour les valeurs positives et négatives, respectivement).
	'p_sep_by_space/n_sep_by_space'	Si le symbole monétaire est séparé de la valeur par une espace ou non (pour les valeurs positives et négatives, respectivement).
	'mon_decimal_point'	Séparateur décimal (entre la partie entière et la partie décimale) utilisé pour les valeurs monétaires.
	'frac_digits'	Nombre de décimales utilisées dans le format local des valeurs monétaires.
	'int_frac_digits'	Nombre de décimales utilisées dans le format international des valeurs monétaires.
	'mon_thousands_sep'	Séparateur de groupe utilisé pour les valeurs monétaires.
	'mon_grouping'	Équivalent de 'grouping', utilisé pour les valeurs monétaires.
	'positive_sign'	Symbole utilisé pour indiquer qu'une valeur monétaire est positive.
	'negative_sign'	Symbole utilisé pour indiquer qu'une valeur monétaire est négative.
	'p_sign_posn/n_sign_posn'	Position du signe (pour les valeurs positives et négatives, respectivement), voir ci-dessous.

Toutes les valeurs numériques peuvent être définies à *CHAR_MAX* pour indiquer qu'il n'y a pas de valeur spécifiée pour ces paramètres régionaux.

Les valeurs possibles pour 'p_sign_posn' et 'n_sign_posn' sont données ci-dessous.

Valeur	Explication
0	Le symbole monétaire et la valeur sont entourés de parenthèses.
1	Le signe doit précéder la valeur et le symbole monétaire.
2	Le signe doit suivre la valeur et le symbole monétaire.
3	Le signe doit précéder immédiatement la valeur.
4	Le signe doit suivre immédiatement la valeur.
CHAR_MAX	Rien n'est spécifié dans ces paramètres régionaux.

The function temporarily sets the `LC_CTYPE` locale to the `LC_NUMERIC` locale or the `LC_MONETARY` locale if locales are different and numeric or monetary strings are non-ASCII. This temporary change affects other threads.

Modifié dans la version 3.7 : The function now temporarily sets the `LC_CTYPE` locale to the `LC_NUMERIC` locale in some cases.

`locale.`**`nl_langinfo`**(*option*)

Renvoie quelques informations spécifiques aux paramètres régionaux sous forme de chaîne. Cette fonction n'est pas disponible sur tous les systèmes et l'ensemble des options possibles peut également varier d'une plateforme à l'autre. Les valeurs possibles pour les arguments sont des nombres, pour lesquels des constantes symboliques sont disponibles dans le module *locale*.

La fonction `nl_langinfo()` accepte l'une des clés suivantes. La plupart des descriptions sont extraites des descriptions correspondantes dans la bibliothèque GNU C.

`locale.`**`CODESET`**

Récupère une chaîne avec le nom de l'encodage des caractères utilisé par le paramètre régional sélectionné.

`locale.`**`D_T_FMT`**

Récupère une chaîne qui peut être utilisée comme une chaîne de format par `time.strftime()` afin de représenter la date et l'heure pour un paramètre régional spécifique.

`locale.`**`D_FMT`**

Récupère une chaîne qui peut être utilisée comme une chaîne de format par `time.strftime()` afin de représenter une date pour un paramètre régional spécifique.

`locale.`**`T_FMT`**

Récupère une chaîne qui peut être utilisée comme une chaîne de format par `time.strftime()` afin de représenter une heure pour un paramètre régional spécifique.

`locale.`**`T_FMT_AMPM`**

Récupère une chaîne de format pour `time.strftime()` afin de représenter l'heure au format am / pm.

`locale.`**`DAY_1`**

`locale.`**`DAY_2`**

`locale.`**`DAY_3`**

`locale.`**`DAY_4`**

`locale.`**`DAY_5`**

`locale.`**`DAY_6`**

`locale.`**`DAY_7`**

Récupère le nom du n-ième jour de la semaine.

> **Note**
>
> Cela suit la convention américaine qui définit `DAY_1` comme étant dimanche, et non la convention internationale (ISO 8601) où lundi est le premier jour de la semaine.

`locale.`**`ABDAY_1`**

`locale.`**`ABDAY_2`**

`locale.`**`ABDAY_3`**

`locale.`**`ABDAY_4`**

`locale.`**`ABDAY_5`**

`locale.`**`ABDAY_6`**

`locale.`**`ABDAY_7`**

Récupère l'abréviation du n-ième jour de la semaine.

locale.**MON_1**
locale.**MON_2**
locale.**MON_3**
locale.**MON_4**
locale.**MON_5**
locale.**MON_6**
locale.**MON_7**
locale.**MON_8**
locale.**MON_9**
locale.**MON_10**
locale.**MON_11**
locale.**MON_12**

Récupère le nom du n-ième mois.

locale.**ABMON_1**
locale.**ABMON_2**
locale.**ABMON_3**
locale.**ABMON_4**
locale.**ABMON_5**
locale.**ABMON_6**
locale.**ABMON_7**
locale.**ABMON_8**
locale.**ABMON_9**
locale.**ABMON_10**
locale.**ABMON_11**
locale.**ABMON_12**

Récupère l'abréviation du n-ième mois.

locale.**RADIXCHAR**

Récupère le caractère de séparation *radix* (point décimal, virgule décimale, etc.).

locale.**THOUSEP**

Récupère le caractère de séparation des milliers (groupes de 3 chiffres).

locale.**YESEXPR**

Récupère une expression régulière qui peut être utilisée par la fonction *regex* pour reconnaître une réponse positive à une question fermée (oui / non).

locale.**NOEXPR**

Get a regular expression that can be used with the `regex(3)` function to recognize a negative response to a yes/no question.

> **Note**
>
> The regular expressions for *YESEXPR* and *NOEXPR* use syntax suitable for the `regex` function from the C library, which might differ from the syntax used in *re*.

locale.**CRNCYSTR**

Récupère le symbole monétaire, précédé de « - » si le symbole doit apparaître avant la valeur, « + » s'il doit apparaître après la valeur, ou « . » s'il doit remplacer le caractère de séparation *radix*.

locale.**ERA**

Get a string which describes how years are counted and displayed for each era in a locale.

La plupart des paramètres régionaux ne définissent pas cette valeur. Un exemple de région qui définit bien cette valeur est le japonais. Au Japon, la représentation traditionnelle des dates comprend le nom de l'ère correspondant au règne de l'empereur de l'époque.

Normally it should not be necessary to use this value directly. Specifying the `E` modifier in their format strings causes the `time.strftime()` function to use this information. The format of the returned string is specified in *The Open Group Base Specifications Issue 8*, paragraph 7.3.5.2 LC_TIME C-Language Access.

locale.**ERA_D_T_FMT**

Récupère la chaîne de format pour `time.strftime()` afin de représenter la date et l'heure pour un paramètre régional spécifique basée sur une ère.

locale.**ERA_D_FMT**

Récupère la chaîne de format pour `time.strftime()` afin de représenter une date pour un paramètre régional spécifique basée sur une ère.

locale.**ERA_T_FMT**

Récupère la chaîne de format pour `time.strftime()` afin de représenter une heure pour un paramètre régional spécifique basée sur une ère.

locale.**ALT_DIGITS**

Get a string consisting of up to 100 semicolon-separated symbols used to represent the values 0 to 99 in a locale-specific way. In most locales this is an empty string.

The function temporarily sets the `LC_CTYPE` locale to the locale of the category that determines the requested value (`LC_TIME`, `LC_NUMERIC`, `LC_MONETARY` or `LC_MESSAGES`) if locales are different and the resulting string is non-ASCII. This temporary change affects other threads.

Modifié dans la version 3.14 : The function now temporarily sets the `LC_CTYPE` locale in some cases.

locale.**getdefaultlocale**([*envvars*])

Tente de déterminer les paramètres régionaux par défaut, puis les renvoie sous la forme d'un n-uplet `(code de la langue, encodage)`.

D'après POSIX, un programme qui n'a pas appelé `setlocale(LC_ALL, '')` fonctionne en utilisant le paramètre régional portable `'C'`. Appeler `setlocale(LC_ALL, '')` lui permet d'utiliser les paramètres régionaux par défaut définis par la variable `LANG`. Comme nous ne voulons pas interférer avec les paramètres régionaux actuels, nous émulons donc le comportement décrit ci-dessus.

Afin de maintenir la compatibilité avec d'autres plateformes, non seulement la variable `LANG` est testée, mais c'est aussi le cas pour toute une liste de variables passés en paramètre via *envvars*. La première variable à être définie sera utilisée. *envvars* utilise par défaut le chemin de recherche utilisé dans GNU *gettext* ; il doit toujours contenir le nom de variable `'LANG'`. Le chemin de recherche de GNU *gettext* contient `'LC_ALL'`, `'LC_CTYPE'`, `'LANG'` et `'LANGUAGE'`, dans cet ordre.

À l'exception du code `'C'`, le code d'une langue correspond à la **RFC 1766**. Le *code de la langue* et l'*encodage* peuvent valoir `None` si leur valeur ne peut être déterminée.

Deprecated since version 3.11, will be removed in version 3.15.

locale.**getlocale**(*category=LC_CTYPE*)

Returns the current setting for the given locale category as sequence containing *language code*, *encoding*. *category* may be one of the `LC_*` values except `LC_ALL`. It defaults to `LC_CTYPE`.

À l'exception du code `'C'`, le code d'une langue correspond à la **RFC 1766**. Le *code de la langue* et l'*encodage* peuvent valoir `None` si leur valeur ne peut être déterminée.

locale.**getpreferredencoding**(*do_setlocale=True*)

Return the *locale encoding* used for text data, according to user preferences. User preferences are expressed differently on different systems, and might not be available programmatically on some systems, so this function only returns a guess.

Sur certains systèmes, il est nécessaire d'invoquer `setlocale()` pour obtenir les préférences de l'utilisateur, cette fonction n'est donc pas utilisable sans protection dans les programmes à fils d'exécutions multiples. Si l'appel de `setlocale` n'est pas nécessaire ou souhaité, *do_setlocale* doit être réglé à `False`.

On Android or if the *Python UTF-8 Mode* is enabled, always return `'utf-8'`, the *locale encoding* and the *do_setlocale* argument are ignored.

The Python preinitialization configures the LC_CTYPE locale. See also the *filesystem encoding and error handler*.

Modifié dans la version 3.7 : The function now always returns `"utf-8"` on Android or if the *Python UTF-8 Mode* is enabled.

`locale.`**`getencoding`**`()`

Get the current *locale encoding* :

— On Android and VxWorks, return `"utf-8"`.

— On Unix, return the encoding of the current `LC_CTYPE` locale. Return `"utf-8"` if `nl_langinfo(CODESET)` returns an empty string : for example, if the current LC_CTYPE locale is not supported.

— On Windows, return the ANSI code page.

The Python preinitialization configures the LC_CTYPE locale. See also the *filesystem encoding and error handler*.

This function is similar to `getpreferredencoding(False)` except this function ignores the *Python UTF-8 Mode*.

Ajouté dans la version 3.11.

`locale.`**`normalize`**`(`*localename*`)`

Renvoie un code normalisé pour le nom du paramètre régional fourni. Ce code renvoyé est structuré de façon à être utilisé avec `setlocale()`. Si la normalisation échoue, le nom d'origine est renvoyé inchangé.

Si l'encodage donné n'est pas connu, la fonction utilise l'encodage par défaut pour le code du paramètre régional, tout comme `setlocale()`.

`locale.`**`strcoll`**`(`*string1*, *string2*`)`

Compare deux chaînes en se basant sur le paramètre `LC_COLLATE` actuel. Comme toute autre fonction de comparaison, renvoie une valeur négative, positive, ou `0`, selon si *string1* est lexicographiquement inférieure, supérieure, ou égale à *string2*.

`locale.`**`strxfrm`**`(`*string*`)`

Transforme une chaîne de caractères en une chaîne qui peut être utilisée dans les comparaisons sensibles aux paramètres régionaux. Par exemple, `strxfrm(s1) < strxfrm(s2)` est équivalent à `strcoll(s1, s2) < 0`. Cette fonction peut être utilisée lorsque la même chaîne est comparée de façon répétitive, par exemple lors de l'assemblage d'une séquence de chaînes.

`locale.`**`format_string`**`(`*format*, *val*, *grouping=False*, *monetary=False*`)`

Formats a number *val* according to the current `LC_NUMERIC` setting. The format follows the conventions of the `%` operator. For floating-point values, the decimal point is modified if appropriate. If *grouping* is `True`, also takes the grouping into account.

Si *monetary* est vrai, la conversion utilise un séparateur des milliers monétaire et des chaînes de regroupement.

Traite les marqueurs de structure en `format % val`, mais en prenant en compte les paramètres régionaux actuels.

Modifié dans la version 3.7 : The *monetary* keyword parameter was added.

`locale.`**`currency`**`(`*val*, *symbol=True*, *grouping=False*, *international=False*`)`

Structure un nombre *val* en fonction du paramètre `LC_MONETARY` actuel.

The returned string includes the currency symbol if *symbol* is true, which is the default. If *grouping* is `True` (which is not the default), grouping is done with the value. If *international* is `True` (which is not the default), the international currency symbol is used.

> **Note**
>
> This function will not work with the 'C' locale, so you have to set a locale via `setlocale()` first.

`locale.str(float)`

Formats a floating-point number using the same format as the built-in function `str(float)`, but takes the decimal point into account.

`locale.delocalize(string)`

Convertit une chaîne de caractères en une chaîne de nombres normalisés, en suivant les réglages `LC_NUMERIC`.

Ajouté dans la version 3.5.

`locale.localize(string, grouping=False, monetary=False)`

Converts a normalized number string into a formatted string following the `LC_NUMERIC` settings.

Ajouté dans la version 3.10.

`locale.atof(string, func=float)`

Converts a string to a number, following the `LC_NUMERIC` settings, by calling *func* on the result of calling `delocalize()` on *string*.

`locale.atoi(string)`

Convertit une chaîne de caractères en un entier, en suivant les réglages `LC_NUMERIC`.

`locale.LC_CTYPE`

Locale category for the character type functions. Most importantly, this category defines the text encoding, i.e. how bytes are interpreted as Unicode codepoints. See **PEP 538** and **PEP 540** for how this variable might be automatically coerced to `C.UTF-8` to avoid issues created by invalid settings in containers or incompatible settings passed over remote SSH connections.

Python doesn't internally use locale-dependent character transformation functions from `ctype.h`. Instead, an internal `pyctype.h` provides locale-independent equivalents like `Py_TOLOWER`.

`locale.LC_COLLATE`

Catégorie de paramètre régional pour les tris de chaînes de caractères. Les fonctions `strcoll()` et `strxfrm()` du module `locale` sont concernées.

`locale.LC_TIME`

Catégorie de paramètre régional pour la mise en forme de la date et de l'heure. La fonction `time.strftime()` suit ces conventions.

`locale.LC_MONETARY`

Catégorie de paramètre régional pour la mise en forme des valeurs monétaires. Les options disponibles sont accessibles à partir de la fonction `localeconv()`.

`locale.LC_MESSAGES`

Catégorie de paramètre régional pour l'affichage de messages. Actuellement, Python ne gère pas les messages spécifiques aux applications qui sont sensibles aux paramètres régionaux. Les messages affichés par le système d'exploitation, comme ceux renvoyés par `os.strerror()` peuvent être affectés par cette catégorie.

This value may not be available on operating systems not conforming to the POSIX standard, most notably Windows.

`locale.LC_NUMERIC`

Locale category for formatting numbers. The functions `format_string()`, `atoi()`, `atof()` and `str()` of the `locale` module are affected by that category. All other numeric formatting operations are not affected.

`locale.LC_ALL`

Combinaison de tous les paramètres régionaux. Si cette option est utilisée lors du changement de paramètres régionaux, la définition de ces paramètres pour toutes les catégories est tentée. Si cela échoue pour n'importe quelle catégorie, aucune d'entre elles n'est modifiée. Lorsque les paramètres régionaux sont récupérés à l'aide

de cette option, une chaîne de caractères indiquant le réglage pour toutes les catégories est renvoyée. Cette chaîne peut alors être utilisée plus tard pour restaurer les paramètres d'origine.

`locale.CHAR_MAX`

Ceci est une constante symbolique utilisée pour différentes valeurs renvoyées par `localeconv()`.

Exemple :

```
>>> import locale
>>> loc = locale.getlocale()  # get current locale
# use German locale; name might vary with platform
>>> locale.setlocale(locale.LC_ALL, 'de_DE')
>>> locale.strcoll('f\xe4n', 'foo')  # compare a string containing an umlaut
>>> locale.setlocale(locale.LC_ALL, '')   # use user's preferred locale
>>> locale.setlocale(locale.LC_ALL, 'C')  # use default (C) locale
>>> locale.setlocale(locale.LC_ALL, loc)  # restore saved locale
```

24.2.1 Contexte, détails, conseils, astuces et mises en garde

The C standard defines the locale as a program-wide property that may be relatively expensive to change. On top of that, some implementations are broken in such a way that frequent locale changes may cause core dumps. This makes the locale somewhat painful to use correctly.

Initialement, lorsqu'un programme est démarré, les paramètres régionaux `C` sont utilisés, peu importe les réglages de l'utilisateur. Il y a toutefois une exception : la catégorie `LC_CTYPE` est modifiée au démarrage pour définir l'encodage des paramètres régionaux actuels comme celui défini par l'utilisateur. Le programme doit explicitement dire qu'il veut utiliser les réglages de l'utilisateur pour les autres catégories, en appelant `setlocale(LC_ALL, '')`.

C'est généralement une mauvaise idée d'appeler `setlocale()` dans une routine de bibliothèque car cela a pour effet secondaire d'affecter le programme entier. Sauvegarder et restaurer les paramètres est presque aussi mauvais : c'est coûteux et cela affecte d'autres fils d'exécutions qui s'exécutent avant que les paramètres n'aient été restaurés.

Si, lors du développement d'un module à usage général, vous avez besoin d'une version indépendante des paramètres régionaux pour une opération y étant sensible (comme c'est le cas pour certains formats utilisés avec `time.strftime()`), vous devez trouver un moyen de le faire sans utiliser la routine de la bibliothèque standard. Le mieux est encore de se convaincre que l'usage des paramètres régionaux est une bonne chose. Ce n'est qu'en dernier recours que vous devez documenter que votre module n'est pas compatible avec les réglages du paramètre régional `C`.

The only way to perform numeric operations according to the locale is to use the special functions defined by this module : `atof()`, `atoi()`, `format_string()`, `str()`.

Il n'y a aucun moyen d'effectuer des conversions de casse et des classifications de caractères en fonction des paramètres régionaux. Pour les chaînes de caractères (Unicode), celles-ci se font uniquement en fonction de la valeur du caractère, tandis que pour les chaînes d'octets, les conversions et les classifications se font en fonction de la valeur ASCII de l'octet, et les octets dont le bit de poids fort est à 1 (c'est-à-dire les octets non ASCII) ne sont jamais convertis ou considérés comme faisant partie d'une classe de caractères comme une lettre ou une espace.

24.2.2 Pour les auteurs d'extensions et les programmes qui intègrent Python

Les modules d'extensions ne devraient jamais appeler `setlocale()`, sauf pour connaître le paramètre régional actuel. Mais comme la valeur renvoyée ne peut être utilisée que pour le restaurer, ce n'est pas très utile (sauf peut-être pour savoir si le paramètre régional est défini à `C` ou non).

When Python code uses the `locale` module to change the locale, this also affects the embedding application. If the embedding application doesn't want this to happen, it should remove the `_locale` extension module (which does all the work) from the table of built-in modules in the `config.c` file, and make sure that the `_locale` module is not accessible as a shared library.

24.2.3 Accéder aux catalogues de messages

`locale.gettext(msg)`

`locale.dgettext(domain, msg)`

`locale.dcgettext(domain, msg, category)`

`locale.textdomain(domain)`

`locale.bindtextdomain(domain, dir)`

`locale.bind_textdomain_codeset(domain, codeset)`

The locale module exposes the C library's gettext interface on systems that provide this interface. It consists of the functions `gettext()`, `dgettext()`, `dcgettext()`, `textdomain()`, `bindtextdomain()`, and `bind_textdomain_codeset()`. These are similar to the same functions in the `gettext` module, but use the C library's binary format for message catalogs, and the C library's search algorithms for locating message catalogs.

Python applications should normally find no need to invoke these functions, and should use `gettext` instead. A known exception to this rule are applications that link with additional C libraries which internally invoke C functions `gettext` or `dcgettext`. For these applications, it may be necessary to bind the text domain, so that the libraries can properly locate their message catalogs.

CHAPITRE 25

Cadriciels d'applications

Les modules décrits dans ce chapitre sont des cadriciels (*frameworks*, en anglais) qui encadreront la structure de vos programmes. Actuellement tous les modules décrits ici sont destinés à écrire des interfaces en ligne de commande.

La liste complète des modules décrits dans ce chapitre est :

25.1 `turtle` — Tortue graphique

Code Source : Lib/turtle.py

25.1.1 Introduction

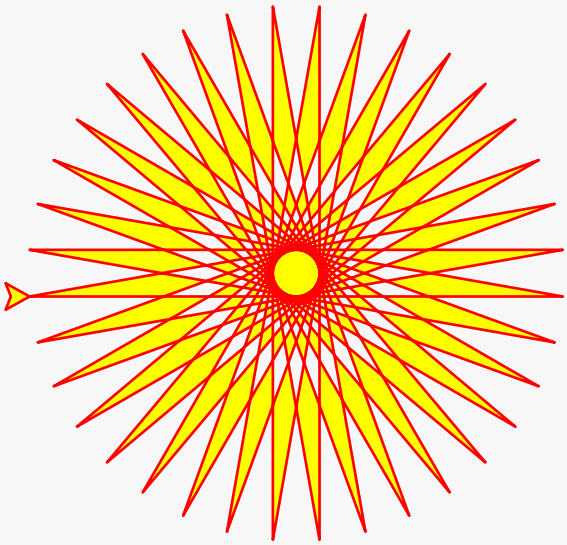
Turtle graphics is an implementation of the popular geometric drawing tools introduced in Logo, developed by Wally Feurzeig, Seymour Papert and Cynthia Solomon in 1967.

25.1.2 Get started

Imaginez un robot sous forme de tortue partant au centre (0, 0) d'un plan cartésien x-y. Après un `import turtle`, exécutez la commande `turtle.forward(15)` et la tortue se déplace (sur l'écran) de 15 pixels en face d'elle, en dessinant une ligne.

Turtle star

La tortue permet de dessiner des formes complexes en utilisant un programme qui répète des actions élémentaires.



In Python, turtle graphics provides a representation of a physical "turtle" (a little robot with a pen) that draws on a sheet of paper on the floor.

It's an effective and well-proven way for learners to encounter programming concepts and interaction with software, as it provides instant, visible feedback. It also provides convenient access to graphical output in general.

Turtle drawing was originally created as an educational tool, to be used by teachers in the classroom. For the programmer who needs to produce some graphical output it can be a way to do that without the overhead of introducing more complex or external libraries into their work.

25.1.3 Tutoriel

New users should start here. In this tutorial we'll explore some of the basics of turtle drawing.

Starting a turtle environment

In a Python shell, import all the objects of the `turtle` module :

```
from turtle import *
```

If you run into a `No module named '_tkinter'` error, you'll have to install the *Tk interface package* on your system.

Basic drawing

Send the turtle forward 100 steps :

```
forward(100)
```

You should see (most likely, in a new window on your display) a line drawn by the turtle, heading East. Change the direction of the turtle, so that it turns 120 degrees left (anti-clockwise) :

```
left(120)
```

Let's continue by drawing a triangle :

```
forward(100)
left(120)
forward(100)
```

Notice how the turtle, represented by an arrow, points in different directions as you steer it.

Experiment with those commands, and also with `backward()` and `right()`.

Réglage des stylos

Try changing the color - for example, `color('blue')` - and width of the line - for example, `width(3)` - and then drawing again.

You can also move the turtle around without drawing, by lifting up the pen : `up()` before moving. To start drawing again, use `down()`.

The turtle's position

Send your turtle back to its starting-point (useful if it has disappeared off-screen) :

```
home()
```

The home position is at the center of the turtle's screen. If you ever need to know them, get the turtle's x-y coordinates with :

```
pos()
```

Home is at `(0, 0)`.

And after a while, it will probably help to clear the window so we can start anew :

```
clearscreen()
```

Making algorithmic patterns

Using loops, it's possible to build up geometric patterns :

```
for steps in range(100):
    for c in ('blue', 'red', 'green'):
        color(c)
        forward(steps)
        right(30)
```

- which of course, are limited only by the imagination !

Let's draw the star shape at the top of this page. We want red lines, filled in with yellow :

```
color('red')
fillcolor('yellow')
```

Just as `up()` and `down()` determine whether lines will be drawn, filling can be turned on and off :

```
begin_fill()
```

Next we'll create a loop :

```
while True:
    forward(200)
    left(170)
    if abs(pos()) < 1:
        break
```

`abs(pos()) < 1` is a good way to know when the turtle is back at its home position.

Finally, complete the filling :

```
end_fill()
```

(Note that filling only actually takes place when you give the `end_fill()` command.)

25.1.4 How to...

This section covers some typical turtle use-cases and approaches.

Get started as quickly as possible

One of the joys of turtle graphics is the immediate, visual feedback that's available from simple commands - it's an excellent way to introduce children to programming ideas, with a minimum of overhead (not just children, of course).

The turtle module makes this possible by exposing all its basic functionality as functions, available with `from turtle import *`. The *turtle graphics tutorial* covers this approach.

It's worth noting that many of the turtle commands also have even more terse equivalents, such as `fd()` for *`forward()`*. These are especially useful when working with learners for whom typing is not a skill.

> You'll need to have the *`Tk interface package`* installed on your system for turtle graphics to work. Be warned that this is not always straightforward, so check this in advance if you're planning to use turtle graphics with a learner.

Automatically begin and end filling

Starting with Python 3.14, you can use the *`fill()` context manager* instead of *`begin_fill()`* and *`end_fill()`* to automatically begin and end fill. Here is an example :

```
with fill():
    for i in range(4):
        forward(100)
        right(90)

forward(200)
```

The code above is equivalent to :

```
begin_fill()
for i in range(4):
    forward(100)
    right(90)
end_fill()

forward(200)
```

Use the `turtle` module namespace

Using `from turtle import *` is convenient - but be warned that it imports a rather large collection of objects, and if you're doing anything but turtle graphics you run the risk of a name conflict (this becomes even more an issue if you're using turtle graphics in a script where other modules might be imported).

The solution is to use `import turtle` - `fd()` becomes `turtle.fd()`, `width()` becomes `turtle.width()` and so on. (If typing "turtle" over and over again becomes tedious, use for example `import turtle as t` instead.)

Use turtle graphics in a script

It's recommended to use the `turtle` module namespace as described immediately above, for example :

```
import turtle as t
from random import random

for i in range(100):
    steps = int(random() * 100)
    angle = int(random() * 360)
    t.right(angle)
    t.fd(steps)
```

Another step is also required though - as soon as the script ends, Python will also close the turtle's window. Add :

```
t.mainloop()
```

to the end of the script. The script will now wait to be dismissed and will not exit until it is terminated, for example by closing the turtle graphics window.

Use object-oriented turtle graphics

Voir aussi

Explanation of the object-oriented interface

Other than for very basic introductory purposes, or for trying things out as quickly as possible, it's more usual and much more powerful to use the object-oriented approach to turtle graphics. For example, this allows multiple turtles on screen at once.

In this approach, the various turtle commands are methods of objects (mostly of `Turtle` objects). You *can* use the object-oriented approach in the shell, but it would be more typical in a Python script.

The example above then becomes :

```
from turtle import Turtle
from random import random

t = Turtle()
for i in range(100):
    steps = int(random() * 100)
    angle = int(random() * 360)
    t.right(angle)
    t.fd(steps)

t.screen.mainloop()
```

Note the last line. `t.screen` is an instance of the *`Screen`* that a Turtle instance exists on ; it's created automatically along with the turtle.

The turtle's screen can be customised, for example :

```
t.screen.title('Object-oriented turtle demo')
t.screen.bgcolor("orange")
```

25.1.5 Turtle graphics reference

Note

La liste des paramètres des fonctions est donnée dans cette documentation. Les méthodes ont, évidemment, le paramètre *self* comme premier argument, mais ce dernier n'est pas indiqué ici.

Les méthodes du module *Turtle*

Les mouvements dans le module *Turtle*

Bouger et dessiner

forward() | *fd()*
backward() | *bk()* | *back()*
right() | *rt()*
left() | *lt()*
goto() | *setpos()* | *setposition()*
teleport()
setx()
sety()
setheading() | *seth()*
home()
circle()
dot()
stamp()
clearstamp()
clearstamps()
undo()
speed()

Connaître l'état de la tortue

position() | *pos()*
towards()
xcor()
ycor()
heading()
distance()

Paramétrage et mesure

degrees()
radians()

Réglage des stylos

État des stylos

pendown() | *pd()* | *down()*
penup() | *pu()* | *up()*
pensize() | *width()*
pen()
isdown()

Réglage des couleurs

color()
pencolor()
fillcolor()

Remplissage

filling()
fill()
begin_fill()
end_fill()

Plus des réglages pour le dessin

reset()

clear()
write()

État de la tortue

Visibilité

showturtle() | *st()*
hideturtle() | *ht()*
isvisible()

Apparence

shape()
resizemode()
shapesize() | *turtlesize()*
shearfactor()
tiltangle()
tilt()
shapetransform()
get_shapepoly()

Utilisation des événements

onclick()
onrelease()
ondrag()

Méthodes spéciales de la tortue

poly()
begin_poly()
end_poly()
get_poly()
clone()
getturtle() | *getpen()*
getscreen()
setundobuffer()
undobufferentries()

Méthodes de *TurtleScreen*/*Screen*

Réglage de la fenêtre

bgcolor()
bgpic()
clearscreen()
resetscreen()
screensize()
setworldcoordinates()

Réglage de l'animation

no_animation()
delay()
tracer()
update()

Utilisation des événements concernant l'écran

listen()
onkey() | *onkeyrelease()*
onkeypress()

onclick() | *onscreenclick()*
ontimer()
mainloop() | *done()*

Paramétrages et méthodes spéciales

mode()
colormode()
getcanvas()
getshapes()
register_shape() | *addshape()*
turtles()
window_height()
window_width()

Méthodes de saisie

textinput()
numinput()

Méthodes spécifiques de *Screen*

bye()
exitonclick()
save()
setup()
title()

25.1.6 Méthodes de *RawTurtle*/*Turtle* et leurs fonctions correspondantes

La plupart des exemples de cette section se réfèrent à une instance de *Turtle* appelée `turtle`.

Les mouvements dans le module *Turtle*

`turtle.`**`forward`**(*distance*)
`turtle.`**`fd`**(*distance*)

Paramètres
`distance` -- un nombre (entier ou flottant)

Avance la tortue de la *distance* spécifiée, dans la direction où elle se dirige.

```
>>> turtle.position()
(0.00,0.00)
>>> turtle.forward(25)
>>> turtle.position()
(25.00,0.00)
>>> turtle.forward(-75)
>>> turtle.position()
(-50.00,0.00)
```

`turtle.`**`back`**(*distance*)
`turtle.`**`bk`**(*distance*)
`turtle.`**`backward`**(*distance*)

Paramètres
`distance` -- un nombre

Déplace la tortue de *distance* vers l'arrière (dans le sens opposé à celui vers lequel elle pointe). Ne change pas le cap de la tortue.

```
>>> turtle.position()
(0.00,0.00)
>>> turtle.backward(30)
>>> turtle.position()
(-30.00,0.00)
```

turtle.**right**(*angle*)

turtle.**rt**(*angle*)

Paramètres

angle -- un nombre (entier ou flottant)

Tourne la tortue à droite de *angle* unités (les unités sont par défaut des degrés, mais peuvent être définies via les fonctions `degrees()` et `radians()`). L'orientation de l'angle dépend du mode de la tortue, voir `mode()`.

```
>>> turtle.heading()
22.0
>>> turtle.right(45)
>>> turtle.heading()
337.0
```

turtle.**left**(*angle*)

turtle.**lt**(*angle*)

Paramètres

angle -- un nombre (entier ou flottant)

Tourne la tortue à gauche d'une valeur de *angle* unités (les unités sont par défaut des degrés, mais peuvent être définies via les fonctions `degrees()` et `radians()`). L'orientation de l'angle dépend du mode de la tortue, voir `mode()`.

```
>>> turtle.heading()
22.0
>>> turtle.left(45)
>>> turtle.heading()
67.0
```

turtle.**goto**(*x*, *y=None*)

turtle.**setpos**(*x*, *y=None*)

turtle.**setposition**(*x*, *y=None*)

Paramètres

— **x** -- un nombre ou une paire / un vecteur de nombres

— **y** -- un nombre ou `None`

Si *y* est `None`, *x* doit être une paire de coordonnées, ou bien une instance de `Vec2D` (par exemple, tel que renvoyé par `pos()`).

Déplace la tortue vers une position absolue. Si le stylo est en bas, trace une ligne. Ne change pas l'orientation de la tortue.

```
>>> tp = turtle.pos()
>>> tp
(0.00,0.00)
>>> turtle.setpos(60,30)
>>> turtle.pos()
(60.00,30.00)
>>> turtle.setpos((20,80))
>>> turtle.pos()
(20.00,80.00)
>>> turtle.setpos(tp)
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> turtle.pos()
(0.00,0.00)
```

turtle.**teleport**(*x*, *y=None*, *, *fill_gap=False*)

Paramètres

— **x** -- un nombre ou `None`
— **y** -- un nombre ou `None`
— **fill_gap** -- a boolean

Move turtle to an absolute position. Unlike goto(x, y), a line will not be drawn. The turtle's orientation does not change. If currently filling, the polygon(s) teleported from will be filled after leaving, and filling will begin again after teleporting. This can be disabled with fill_gap=True, which makes the imaginary line traveled during teleporting act as a fill barrier like in goto(x, y).

```
>>> tp = turtle.pos()
>>> tp
(0.00,0.00)
>>> turtle.teleport(60)
>>> turtle.pos()
(60.00,0.00)
>>> turtle.teleport(y=10)
>>> turtle.pos()
(60.00,10.00)
>>> turtle.teleport(20, 30)
>>> turtle.pos()
(20.00,30.00)
```

Ajouté dans la version 3.12.

turtle.**setx**(*x*)

Paramètres

x -- un nombre (entier ou flottant)

Définit la première coordonnée de la tortue à *x*, en laissant la deuxième coordonnée inchangée.

```
>>> turtle.position()
(0.00,240.00)
>>> turtle.setx(10)
>>> turtle.position()
(10.00,240.00)
```

turtle.**sety**(*y*)

Paramètres

y -- un nombre (entier ou flottant)

Définit la deuxième coordonnée de la tortue à *y*, en laissant la première coordonnée inchangée.

```
>>> turtle.position()
(0.00,40.00)
>>> turtle.sety(-10)
>>> turtle.position()
(0.00,-10.00)
```

turtle.**setheading**(*to_angle*)

turtle.**seth**(*to_angle*)

Paramètres

to_angle -- un nombre (entier ou flottant)

Règle l'orientation de la tortue à la valeur `to_angle`. Voici quelques orientations courantes en degrés :

mode standard	mode logo
0 – Est	0 – Nord
90 – Nord	90 – Est
180 – Ouest	180 – Sud
270 – Sud	270 – Ouest

```
>>> turtle.setheading(90)
>>> turtle.heading()
90.0
```

`turtle.`**`home`**`()`

Déplace la tortue à l'origine — coordonnées (0,0) — et l'oriente à son cap initial (qui dépend du mode, voir *`mode()`*).

```
>>> turtle.heading()
90.0
>>> turtle.position()
(0.00,-10.00)
>>> turtle.home()
>>> turtle.position()
(0.00,0.00)
>>> turtle.heading()
0.0
```

`turtle.`**`circle`**`(`*radius*, *extent=None*, *steps=None*`)`

Paramètres

- **`radius`** -- un nombre
- **`extent`** -- un nombre (ou `None`)
- **`steps`** -- un entier (ou `None`)

Dessine un cercle de rayon *radius*. Le centre se trouve à une distance de *radius* à gauche de la tortue ; l'angle *extent* détermine quelle partie du cercle est dessinée. Si *extent* n'est pas fourni, dessine le cercle en entier. Si *extent* ne correspond pas à un cercle entier, la position actuelle du stylo est donnée par l'un des points d'extrémité de l'arc de cercle. Si la valeur de *radius* est positive, dessine l'arc de cercle dans le sens inverse des aiguilles d'une montre, sinon le dessine dans le sens des aiguilles d'une montre. Enfin, la direction de la tortue peut être modifiée en réglant la valeur de *extent*.

Comme le cercle est approximé par un polygone régulier inscrit, *steps* détermine le nombre de pas à utiliser. Si cette valeur n'est pas donnée, elle sera calculée automatiquement. Elle peut être utilisée pour dessiner des polygones réguliers.

```
>>> turtle.home()
>>> turtle.position()
(0.00,0.00)
>>> turtle.heading()
0.0
>>> turtle.circle(50)
>>> turtle.position()
(-0.00,0.00)
>>> turtle.heading()
0.0
>>> turtle.circle(120, 180)  # draw a semicircle
>>> turtle.position()
(0.00,240.00)
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> turtle.heading()
180.0
```

`turtle.dot(size=None, *color)`

Paramètres

— **size** -- un entier supérieur ou égal à 1 (si fourni)

— **color** -- une chaîne qui désigne une couleur ou un triplet de couleur numérique

Dessine un point circulaire de diamètre *size*, de la couleur *color*. Si le paramètre *size* n'est pas indiqué, utilise la valeur maximum de la taille du stylo plus 4 et de la taille du stylo multiplié par 2.

```
>>> turtle.home()
>>> turtle.dot()
>>> turtle.fd(50); turtle.dot(20, "blue"); turtle.fd(50)
>>> turtle.position()
(100.00,-0.00)
>>> turtle.heading()
0.0
```

`turtle.stamp()`

Tamponne une copie de la forme de la tortue sur le canevas à la position actuelle de la tortue. Renvoie un *stamp_id* pour ce tampon, qui peut être utilisé pour le supprimer en appelant `clearstamp(stamp_id)`.

```
>>> turtle.color("blue")
>>> stamp_id = turtle.stamp()
>>> turtle.fd(50)
```

`turtle.clearstamp(stampid)`

Paramètres

stampid -- un entier, doit être la valeur renvoyée par l'appel précédent de *`stamp()`*

Supprime le tampon dont le *stampid* est donné.

```
>>> turtle.position()
(150.00,-0.00)
>>> turtle.color("blue")
>>> astamp = turtle.stamp()
>>> turtle.fd(50)
>>> turtle.position()
(200.00,-0.00)
>>> turtle.clearstamp(astamp)
>>> turtle.position()
(200.00,-0.00)
```

`turtle.clearstamps(n=None)`

Paramètres

n -- un entier (ou `None`)

Supprime tous, les *n* premiers ou les *n* derniers tampons de la tortue. Si *n* est `None`, supprime tous les tampons, si $n > 0$, supprime les *n* premiers tampons et si $n < 0$, supprime les *n* derniers tampons.

```
>>> for i in range(8):
...     unused_stamp_id = turtle.stamp()
...     turtle.fd(30)
>>> turtle.clearstamps(2)
>>> turtle.clearstamps(-2)
>>> turtle.clearstamps()
```

`turtle.undo()`

Annule la ou les dernières (si répété) actions de la tortue. Le nombre d'annulations disponible est déterminé par la taille de la mémoire tampon d'annulations.

```
>>> for i in range(4):
...     turtle.fd(50); turtle.lt(80)
...
>>> for i in range(8):
...     turtle.undo()
```

`turtle.speed(speed=None)`

Paramètres

speed -- un nombre entier compris dans l'intervalle entre 0 et 10 inclus, ou une chaîne de vitesse (voir ci-dessous)

Règle la vitesse de la tortue à une valeur entière comprise entre 0 et 10 inclus. Si aucun argument n'est donné, renvoie la vitesse actuelle.

Si l'entrée est un nombre supérieur à 10 ou inférieur à 0,5, la vitesse est fixée à 0. Les chaînes de vitesse sont mises en correspondance avec les valeurs de vitesse comme suit :

— `"fastest"` : 0
— `"fast"` : 10
— `"normal"` : 6
— `"slow"` : 3
— `"slowest"` : 1

Les vitesses de 1 à 10 permettent une animation de plus en plus rapide du trait du dessin et de la rotation des tortues.

Attention : *speed* = 0 signifie qu'il n'y a *aucune* animation. *forward*/*back* font sauter la tortue et, de même, *left*/*right* font tourner la tortue instantanément.

```
>>> turtle.speed()
3
>>> turtle.speed('normal')
>>> turtle.speed()
6
>>> turtle.speed(9)
>>> turtle.speed()
9
```

Connaître l'état de la tortue

`turtle.position()`

`turtle.pos()`

Renvoie la position actuelle de la tortue (x,y) (en tant qu'un vecteur `Vec2d`).

```
>>> turtle.pos()
(440.00,-0.00)
```

`turtle.towards(x, y=None)`

Paramètres

— **x** -- un nombre, ou une paire / un vecteur de nombres, ou une instance de tortue
— **y** -- un nombre si *x* est un nombre, sinon `None`

Renvoie l'angle entre l'orientation d'origine et la ligne formée de la position de la tortue à la position spécifiée par (x,y), le vecteur ou l'autre tortue. L'orientation d'origine dépend du mode — "standard"/"world" ou "logo".

```
>>> turtle.goto(10, 10)
>>> turtle.towards(0,0)
225.0
```

turtle.**xcor**()

Renvoie la coordonnée x de la tortue.

```
>>> turtle.home()
>>> turtle.left(50)
>>> turtle.forward(100)
>>> turtle.pos()
(64.28,76.60)
>>> print(round(turtle.xcor(), 5))
64.27876
```

turtle.**ycor**()

Renvoie la coordonnée y de la tortue.

```
>>> turtle.home()
>>> turtle.left(60)
>>> turtle.forward(100)
>>> print(turtle.pos())
(50.00,86.60)
>>> print(round(turtle.ycor(), 5))
86.60254
```

turtle.**heading**()

Renvoie le cap de la tortue (la valeur dépend du mode de la tortue, voir `mode()`).

```
>>> turtle.home()
>>> turtle.left(67)
>>> turtle.heading()
67.0
```

turtle.**distance**(*x*, *y=None*)

Paramètres

— **x** -- un nombre, ou une paire / un vecteur de nombres, ou une instance de tortue

— **y** -- un nombre si *x* est un nombre, sinon `None`

Renvoie la distance entre la tortue et (x,y), le vecteur donné ou l'autre tortue donnée. La valeur est exprimée en unités de pas de tortue.

```
>>> turtle.home()
>>> turtle.distance(30,40)
50.0
>>> turtle.distance((30,40))
50.0
>>> joe = Turtle()
>>> joe.forward(77)
>>> turtle.distance(joe)
77.0
```

Paramètres de mesure

turtle.**degrees**(*fullcircle=360.0*)

Paramètres

fullcircle -- un nombre

Définit les unités de mesure des angles, c.-à-d. fixe le nombre de « degrés » pour un cercle complet. La valeur par défaut est de 360 degrés.

```
>>> turtle.home()
>>> turtle.left(90)
>>> turtle.heading()
90.0

>>> # Change angle measurement unit to grad (also known as gon,
>>> # grade, or gradian and equals 1/100-th of the right angle.)
>>> turtle.degrees(400.0)
>>> turtle.heading()
100.0
>>> turtle.degrees(360)
>>> turtle.heading()
90.0
```

`turtle.radians()`

Règle l'unité de mesure des angles sur radians. Équivalent à `degrees(2*math.pi)`.

```
>>> turtle.home()
>>> turtle.left(90)
>>> turtle.heading()
90.0
>>> turtle.radians()
>>> turtle.heading()
1.5707963267948966
```

Réglage des stylos

État des stylos

`turtle.pendown()`

`turtle.pd()`

`turtle.down()`

Baisse la pointe du stylo — dessine quand il se déplace.

`turtle.penup()`

`turtle.pu()`

`turtle.up()`

Lève la pointe du stylo — pas de dessin quand il se déplace.

`turtle.pensize(width=None)`

`turtle.width(width=None)`

Paramètres

width -- un nombre positif

Règle l'épaisseur de la ligne à *width* ou la renvoie. Si *resizemode* est défini à "auto" et que *turtleshape* (la forme de la tortue) est un polygone, le polygone est dessiné avec cette épaisseur. Si aucun argument n'est passé, la taille actuelle du stylo (*pensize*) est renvoyée.

```
>>> turtle.pensize()
1
>>> turtle.pensize(10)    # from here on lines of width 10 are drawn
```

`turtle.pen(pen=None, **pendict)`

Paramètres

- **pen** -- un dictionnaire avec certaines ou toutes les clés énumérées ci-dessous
- **pendict** -- un ou plusieurs arguments par mots-clés avec les clés suivantes comme mots-clés

Renvoie ou définit les attributs du stylo dans un `"pen-dictionary"` avec les paires clés / valeurs suivantes :
— `"shown"` : `True` / `False`
— `"pendown"` : `True` / `False`
— `"pencolor"` : chaîne de caractères ou triplet désignant la couleur du stylo
— `"fillcolor"` : chaîne de caractères ou triplet pour la couleur de remplissage
— `"pensize"` : nombre positif
— `"speed"` : nombre compris dans intervalle 0 et 10
— `"resizemode"` : `"auto"`, `"user"` ou `"noresize"`
— `"stretchfactor"` : (nombre positif, nombre positif)
— `"outline"` : nombre positif
— `"tilt"` : nombre

Ce dictionnaire peut être utilisé comme argument pour un appel ultérieur à `pen()` pour restaurer l'ancien état du stylo. En outre, un ou plus de ces attributs peuvent est passés en tant qu'arguments nommés. Cela peut être utilisé pour définir plusieurs attributs du stylo en une instruction.

```
>>> turtle.pen(fillcolor="black", pencolor="red", pensize=10)
>>> sorted(turtle.pen().items())
[('fillcolor', 'black'), ('outline', 1), ('pencolor', 'red'),
 ('pendown', True), ('pensize', 10), ('resizemode', 'noresize'),
 ('shearfactor', 0.0), ('shown', True), ('speed', 9),
 ('stretchfactor', (1.0, 1.0)), ('tilt', 0.0)]
>>> penstate=turtle.pen()
>>> turtle.color("yellow", "")
>>> turtle.penup()
>>> sorted(turtle.pen().items())[:3]
[('fillcolor', ''), ('outline', 1), ('pencolor', 'yellow')]
>>> turtle.pen(penstate, fillcolor="green")
>>> sorted(turtle.pen().items())[:3]
[('fillcolor', 'green'), ('outline', 1), ('pencolor', 'red')]
```

`turtle.`**`isdown`**`()`

Renvoie `True` si la pointe du stylo est en bas et `False` si elle est en haut.

```
>>> turtle.penup()
>>> turtle.isdown()
False
>>> turtle.pendown()
>>> turtle.isdown()
True
```

Réglage des couleurs

`turtle.`**`pencolor`**`(`**args*`)`

Renvoie ou règle la couleur du stylo.

Quatre formats d'entrée sont autorisés :

`pencolor()`
Renvoie la couleur du stylo actuelle en tant que chaine de spécification de couleurs ou en tant qu'un *n*-uplet (voir l'exemple). Peut être utilisée comme entrée à un autre appel de *color*/*pencolor*/*fillcolor*.

`pencolor(colorstring)`
Définit la couleur du stylo à `colorstring`, qui est une chaîne de spécification de couleur *Tk*, telle que `"red"`, `"yellow"`, ou `"#33cc8c"`.

`pencolor((r, g, b))`
Définit la couleur du stylo à la couleur RGB représentée par le *n*-uplet de *r*, *g* et *b*. Chacun des *r*, *g* et *b* doit être dans l'intervalle `0..colormode`, où *colormode* est vaut 1.0 ou 255 (voir `colormode()`).

`pencolor(r, g, b)`
Définit la couleur du stylo à la couleur RGB représentée par *r*, *g* et *b*. Chacun des *r*, *g* et *b* doit être dans l'intervalle `0..colormode`.

Si la forme de la tortue est un polygone, le contour de ce polygone est dessiné avec la nouvelle couleur du stylo.

```
>>> colormode()
1.0
>>> turtle.pencolor()
'red'
>>> turtle.pencolor("brown")
>>> turtle.pencolor()
'brown'
>>> tup = (0.2, 0.8, 0.55)
>>> turtle.pencolor(tup)
>>> turtle.pencolor()
(0.2, 0.8, 0.5490196078431373)
>>> colormode(255)
>>> turtle.pencolor()
(51.0, 204.0, 140.0)
>>> turtle.pencolor('#32c18f')
>>> turtle.pencolor()
(50.0, 193.0, 143.0)
```

turtle.**fillcolor**(**args*)

Renvoie ou règle la couleur de remplissage.

Quatre formats d'entrée sont autorisés :

fillcolor()
Renvoie la couleur de remplissage actuelle (*fillcolor*) en tant que chaine de spécification, possiblement en format *n*-uplet (voir l'exemple). Peut être utilisée en entrée pour un autre appel de *color*/*pencolor*/*fillcolor*.

fillcolor(colorstring)
Définit la couleur de remplissage (*fillcolor*) à *colorstring*, qui est une chaine de spécification de couleur *Tk* comme par exemple `"red"`, `"yellow"` ou `"#33cc8c"`.

fillcolor((r, g, b))
Définit la couleur du remplissage (*fillcolor*) à la couleur RGB représentée par le *n*-uplet *r*, *g*, *b*. Chacun des *r*, *g* et *b* doit être dans l'intervalle `0..colormode` où *colormode* vaut 1.0 ou 255 (voir `colormode()`).

fillcolor(r, g, b)
Définit la couleur du remplissage(*fillcolor*) à la couleur RGB représentée par *r*, *g* et *b*. Chacun des *r*, *g* et *b* doit être dans l'intervalle `0..colormode`.

Si la forme de la tortue est un polygone, l'intérieur de ce polygone sera dessiné avec la nouvelle couleur de remplissage.

```
>>> turtle.fillcolor("violet")
>>> turtle.fillcolor()
'violet'
>>> turtle.pencolor()
(50.0, 193.0, 143.0)
>>> turtle.fillcolor((50, 193, 143))  # Integers, not floats
>>> turtle.fillcolor()
(50.0, 193.0, 143.0)
>>> turtle.fillcolor('#ffffff')
>>> turtle.fillcolor()
(255.0, 255.0, 255.0)
```

turtle.**color**(**args*)

Renvoie ou règle la couleur du stylo et la couleur de remplissage.

Plusieurs formats d'entrée sont autorisés. Ils peuvent avoir de zéro jusqu'à trois arguments, employés comme suit :

color()
Renvoie la couleur du stylo actuelle et la couleur de remplissage actuelle sous forme de paire, soit de chaines de spécification de couleur, soit de *n*-uplets comme renvoyés par `pencolor()` et `fillcolor()`.

color(colorstring), color((r,g,b)), color(r,g,b)
Les formats d'entrée sont comme dans *pencolor()*. Définit à la fois la couleur de remplissage et la couleur du stylo à la valeur passée.

color(colorstring1, colorstring2), color((r1,g1,b1), (r2,g2,b2))
Équivalent à `pencolor(colorstring1)` et `fillcolor(colorstring2)` et de manière analogue si un autre format d'entrée est utilisé.

Si la forme de la tortue est un polygone, le contour et l'intérieur de ce polygone sont dessinés avec les nouvelles couleurs.

```
>>> turtle.color("red", "green")
>>> turtle.color()
('red', 'green')
>>> color("#285078", "#a0c8f0")
>>> color()
((40.0, 80.0, 120.0), (160.0, 200.0, 240.0))
```

Voir aussi : la méthode *colormode()* de *Screen*.

Remplissage

`turtle.`**`filling`**`()`

Renvoie l'état de remplissage (`True` signifie en train de faire un remplissage, `False` sinon).

```
>>> turtle.begin_fill()
>>> if turtle.filling():
...     turtle.pensize(5)
... else:
...     turtle.pensize(3)
```

`turtle.`**`fill`**`()`

Fill the shape drawn in the `with turtle.fill():` block.

```
>>> turtle.color("black", "red")
>>> with turtle.fill():
...     turtle.circle(80)
```

Using `fill()` is equivalent to adding the *begin_fill()* before the fill-block and *end_fill()* after the fill-block :

```
>>> turtle.color("black", "red")
>>> turtle.begin_fill()
>>> turtle.circle(80)
>>> turtle.end_fill()
```

Ajouté dans la version 3.14.

`turtle.`**`begin_fill`**`()`

À appeler juste avant de dessiner une forme à remplir.

`turtle.`**`end_fill`**`()`

Remplit la forme dessinée après le dernier appel à *begin_fill()*.

Le remplissage correct des formes complexes (polygones qui se recoupent, plusieurs formes) dépend des primitives graphiques du système d'exploitation, du type et du nombre des chevauchements. Par exemple, l'étoile (*Turtle star* en anglais) ci-dessus peut être entièrement jaune ou comporter quelques régions blanches.

```
>>> turtle.color("black", "red")
>>> turtle.begin_fill()
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> turtle.circle(80)
>>> turtle.end_fill()
```

Plus des réglages pour le dessin

`turtle.reset()`

Supprime les dessins de la tortue de l'écran, recentre la tortue et assigne les variables aux valeurs par défaut.

```
>>> turtle.goto(0,-22)
>>> turtle.left(100)
>>> turtle.position()
(0.00,-22.00)
>>> turtle.heading()
100.0
>>> turtle.reset()
>>> turtle.position()
(0.00,0.00)
>>> turtle.heading()
0.0
```

`turtle.clear()`

Supprime les dessins de la tortue de l'écran. Ne déplace pas la tortue. L'état et la position de la tortue ainsi que les dessins des autres tortues ne sont pas affectés.

`turtle.write(arg, move=False, align='left', font=('Arial', 8, 'normal'))`

Paramètres

- **arg** -- objet à écrire sur le *TurtleScreen*
- **move** -- `True` / `False`
- **align** -- l'une des chaînes de caractères suivantes : `"left"`, `"center"` ou `"right"`
- **font** -- triplet (nom de police, taille de police, type de police)

Écrit du texte - La représentation de la chaîne *arg* - à la position actuelle de la tortue conformément à *align* ("*left*", "*center*" ou "*right*") et en police donnée. Si *move* est `True`, le stylo est déplacé vers le coin inférieur droit du texte. Par défaut, *move* est `False`.

```
>>> turtle.write("Home = ", True, align="center")
>>> turtle.write((0,0), True)
```

État de la tortue

Visibilité

`turtle.hideturtle()`

`turtle.ht()`

Rend la tortue invisible. C'est recommandé lorsque vous êtes en train de faire un dessin complexe, vous observerez alors une accélération notable.

```
>>> turtle.hideturtle()
```

`turtle.showturtle()`

`turtle.st()`

Rend la tortue visible.

```
>>> turtle.showturtle()
```

turtle.**isvisible**()

Renvoie `True` si la tortue est visible, et `False` si elle est cachée.

```
>>> turtle.hideturtle()
>>> turtle.isvisible()
False
>>> turtle.showturtle()
>>> turtle.isvisible()
True
```

Apparence

turtle.**shape**(*name=None*)

Paramètres
name -- une chaîne de caractères qui correspond à un nom de forme valide

La tortue prend la forme *name* donnée, ou, si *name* n'est pas donné, renvoie le nom de la forme actuelle. Le nom *name* donné doit exister dans le dictionnaire de formes de *TurtleScreen*. Initialement, il y a les polygones suivants : "*arrow*", "*turtle*", "*circle*", "*square*", "*triangle*", "*classic*". Pour en apprendre plus sur comment gérer les formes, voir la méthode de *Screen* `register_shape()`.

```
>>> turtle.shape()
'classic'
>>> turtle.shape("turtle")
>>> turtle.shape()
'turtle'
```

turtle.**resizemode**(*rmode=None*)

Paramètres
rmode -- l'une des chaînes suivantes : `"auto"`, `"user"`, `"noresize"`

Définit *resizemode* à l'une des valeurs suivantes : "*auto*", "*user*", "*noresize*". Si "*rmode*" n'est pas donné, renvoie le *resizemode* actuel. Les différents *resizemode* ont les effets suivants :

— *"auto"* : adapte l'apparence de la tortue en fonction de la largeur du stylo (*value of pensize* en anglais).
— *"user"* : adapte l'apparence de la tortue en fonction des valeurs du paramètre d'étirement et de la largeur des contours, déterminés par `shapesize()`.
— *"noresize"* : il n'y a pas de modification de l'apparence de la tortue.

`resizemode("user")` est appelé par `shapesize()` quand utilisé sans arguments.

```
>>> turtle.resizemode()
'noresize'
>>> turtle.resizemode("auto")
>>> turtle.resizemode()
'auto'
```

turtle.**shapesize**(*stretch_wid=None*, *stretch_len=None*, *outline=None*)

turtle.**turtlesize**(*stretch_wid=None*, *stretch_len=None*, *outline=None*)

Paramètres
— **stretch_wid** -- nombre positif
— **stretch_len** -- nombre positif
— **outline** -- nombre positif

Return or set the pen's attributes x/y-stretchfactors and/or outline. Set resizemode to "user". If and only if resizemode is set to "user", the turtle will be displayed stretched according to its stretchfactors : *stretch_wid* is stretchfactor perpendicular to its orientation, *stretch_len* is stretchfactor in direction of its orientation, *outline* determines the width of the shape's outline.

```
>>> turtle.shapesize()
(1.0, 1.0, 1)
>>> turtle.resizemode("user")
>>> turtle.shapesize(5, 5, 12)
>>> turtle.shapesize()
(5, 5, 12)
>>> turtle.shapesize(outline=8)
>>> turtle.shapesize()
(5, 5, 8)
```

`turtle.`**`shearfactor`**(*shear=None*)

Paramètres

`shear` -- un nombre (facultatif)

Définit ou renvoie le paramétrage de cisaillement actuel. Déforme la tortue en fonction du paramètre *shear* donné, qui est la tangente de l'angle de cisaillement. Ne change pas le sens de déplacement de la tortue. Si le paramètre *shear* n'est pas indiqué, renvoie la valeur actuelle du cisaillement, c.-à-d. la valeur de la tangente de l'angle de cisaillement, celui par rapport auquel les lignes parallèles à la direction de la tortue sont cisaillées.

```
>>> turtle.shape("circle")
>>> turtle.shapesize(5,2)
>>> turtle.shearfactor(0.5)
>>> turtle.shearfactor()
0.5
```

`turtle.`**`tilt`**(*angle*)

Paramètres

`angle` -- un nombre

Tourne la forme de la tortue de *angle* depuis son angle d'inclinaison actuel, mais *ne change pas* le cap de la tortue (direction du mouvement).

```
>>> turtle.reset()
>>> turtle.shape("circle")
>>> turtle.shapesize(5,2)
>>> turtle.tilt(30)
>>> turtle.fd(50)
>>> turtle.tilt(30)
>>> turtle.fd(50)
```

`turtle.`**`tiltangle`**(*angle=None*)

Paramètres

`angle` -- un nombre (facultatif)

Définit ou renvoie l'angle d'inclinaison actuel. Si l'angle est donné, la forme de la tortue est tournée pour pointer dans direction spécifiée par l'angle, indépendamment de son angle d'inclinaison actuel. *Ne change pas* le cap de la tortue (direction du mouvement). Si l'angle n'est pas donné, renvoie l'angle d'inclinaison actuel (L'angle entre l'orientation de la forme de la tortue et le cap de la tortue (sa direction de mouvement)).

```
>>> turtle.reset()
>>> turtle.shape("circle")
>>> turtle.shapesize(5,2)
>>> turtle.tilt(45)
>>> turtle.tiltangle()
45.0
```

`turtle.`**`shapetransform`**(*t11=None*, *t12=None*, *t21=None*, *t22=None*)

Paramètres

— **t11** -- un nombre (facultatif)
— **t12** -- un nombre (facultatif)
— **t21** -- un nombre (facultatif)
— **t12** -- un nombre (facultatif)

Définit ou renvoie la matrice de transformation actuelle de la forme de la tortue.

Si aucun élément de la matrice n'est fourni, renvoie la matrice de transformation sous la forme d'un *n*-uplet à 4 éléments. Autrement, définit les éléments donnés et transforme la forme de la tortue conformément à la matrice dont la première ligne est t11, t12 et la deuxième ligne t21, t22. Le déterminant t11 * t22 - t12 * t21 ne doit pas être nul, sinon une erreur est levée. Cela modifie le facteur d'étirement, le facteur de cisaillement et l'angle d'inclinaison en fonction de la matrice donnée.

```
>>> turtle = Turtle()
>>> turtle.shape("square")
>>> turtle.shapesize(4,2)
>>> turtle.shearfactor(-0.5)
>>> turtle.shapetransform()
(4.0, -1.0, -0.0, 2.0)
```

`turtle.`**`get_shapepoly`**`()`

Renvoie la forme actuelle du polygone en *n*-uplet de paires de coordonnées. Vous pouvez l'utiliser afin de définir une nouvelle forme ou en tant que composant pour une forme plus complexe.

```
>>> turtle.shape("square")
>>> turtle.shapetransform(4, -1, 0, 2)
>>> turtle.get_shapepoly()
((50, -20), (30, 20), (-50, 20), (-30, -20))
```

Utilisation des événements

`turtle.`**`onclick`**`(`*fun*, *btn=1*, *add=None*`)`

Paramètres

— **fun** -- une fonction à deux arguments qui sera appelée avec les coordonnées du point cliqué sur le canevas
— **btn** -- numéro du bouton de la souris, par défaut 1 (bouton de gauche)
— **add** -- `True` ou `False` — si `True`, un nouveau lien est ajouté, sinon il remplace un ancien lien

Crée un lien vers *fun* pour les événements de clics de la souris sur cette tortue. Si *fun* est `None`, les liens existants sont supprimés. Exemple pour la tortue anonyme, c'est-à-dire la manière procédurale :

```
>>> def turn(x, y):
...     left(180)
...
>>> onclick(turn)  # Now clicking into the turtle will turn it.
>>> onclick(None)  # event-binding will be removed
```

`turtle.`**`onrelease`**`(`*fun*, *btn=1*, *add=None*`)`

Paramètres

— **fun** -- une fonction à deux arguments qui sera appelée avec les coordonnées du point cliqué sur le canevas
— **btn** -- numéro du bouton de la souris, par défaut 1 (bouton de gauche)
— **add** -- `True` ou `False` — si `True`, un nouveau lien est ajouté, sinon il remplace un ancien lien

Crée un lien vers *fun* pour les événements de relâchement d'un clic de la souris sur cette tortue. Si *fun* est `None`, les liens existants sont supprimés.

```
>>> class MyTurtle(Turtle):
...     def glow(self,x,y):
...         self.fillcolor("red")
...     def unglow(self,x,y):
...         self.fillcolor("")
...
>>> turtle = MyTurtle()
>>> turtle.onclick(turtle.glow)     # clicking on turtle turns fillcolor red,
>>> turtle.onrelease(turtle.unglow) # releasing turns it to transparent.
```

`turtle.`**`ondrag`**(*fun*, *btn=1*, *add=None*)

Paramètres

- **`fun`** -- une fonction à deux arguments qui sera appelée avec les coordonnées du point cliqué sur le canevas
- **`btn`** -- numéro du bouton de la souris, par défaut 1 (bouton de gauche)
- **`add`** -- `True` ou `False` — si `True`, un nouveau lien est ajouté, sinon il remplace un ancien lien

Crée un lien vers *fun* pour les événements de mouvement de la souris sur cette tortue. Si *fun* est `None`, les liens existants sont supprimés.

Remarque : toutes les séquences d'événements de mouvement de la souris sur une tortue sont précédées par un événement de clic de la souris sur cette tortue.

```
>>> turtle.ondrag(turtle.goto)
```

Par la suite, un cliquer-glisser sur la tortue la fait se déplacer au travers de l'écran, produisant ainsi des dessins « à la main » (si le stylo est posé).

Méthodes spéciales de la tortue

`turtle.`**`poly`**()

Record the vertices of a polygon drawn in the `with turtle.poly():` block. The first and last vertices will be connected.

```
>>> with turtle.poly():
...     turtle.forward(100)
...     turtle.right(60)
...     turtle.forward(100)
```

Ajouté dans la version 3.14.

`turtle.`**`begin_poly`**()

Démarre l'enregistrement des sommets d'un polygone. La position actuelle de la tortue est le premier sommet du polygone.

`turtle.`**`end_poly`**()

Arrête l'enregistrement des sommets d'un polygone. La position actuelle de la tortue sera le dernier sommet du polygone. Il sera connecté au premier sommet.

`turtle.`**`get_poly`**()

Renvoie le dernier polygone sauvegardé.

```
>>> turtle.home()
>>> turtle.begin_poly()
>>> turtle.fd(100)
>>> turtle.left(20)
>>> turtle.fd(30)
>>> turtle.left(60)
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> turtle.fd(50)
>>> turtle.end_poly()
>>> p = turtle.get_poly()
>>> register_shape("myFavouriteShape", p)
```

turtle.**clone**()

Crée et renvoie un clone de la tortue avec les mêmes position, cap et propriétés.

```
>>> mick = Turtle()
>>> joe = mick.clone()
```

turtle.**getturtle**()

turtle.**getpen**()

Renvoie l'objet *Turtle* lui-même. Sa seule utilisation : comme fonction pour renvoyer la "tortue anonyme" :

```
>>> pet = getturtle()
>>> pet.fd(50)
>>> pet
<turtle.Turtle object at 0x...>
```

turtle.**getscreen**()

Renvoie l'objet `TurtleScreen` sur lequel la tortue dessine. Les méthodes de TurtleScreen peuvent être appelées pour cet objet.

```
>>> ts = turtle.getscreen()
>>> ts
<turtle._Screen object at 0x...>
>>> ts.bgcolor("pink")
```

turtle.**setundobuffer**(*size*)

Paramètres

size -- un entier ou `None`

Définit ou désactive la mémoire d'annulation. Si *size* est un entier, une mémoire d'annulation de la taille donnée est installée. *size* donne le nombre maximum d'actions de la tortue qui peuvent être annulées par la fonction/méthode `undo()`. Si *size* est `None`, la mémoire d'annulation est désactivée.

```
>>> turtle.setundobuffer(42)
```

turtle.**undobufferentries**()

Renvoie le nombre d'entrées dans la mémoire d'annulation.

```
>>> while undobufferentries():
...     undo()
```

Formes composées

Pour utiliser des formes de tortues combinées, qui sont composées de polygones de différentes couleurs, vous devez utiliser la classe utilitaire `Shape` explicitement comme décrit ci-dessous :

1. Créez un objet Shape vide de type "compound".
2. Add as many components to this object as desired, using the `addcomponent()` method.

   Par exemple :

```
>>> s = Shape("compound")
>>> poly1 = ((0,0),(10,-5),(0,10),(-10,-5))
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> s.addcomponent(poly1, "red", "blue")
>>> poly2 = ((0,0),(10,-5),(-10,-5))
>>> s.addcomponent(poly2, "blue", "red")
```

3. Maintenant ajoutez la *Shape* à la liste des formes de *Screen* et utilisez la :

```
>>> register_shape("myshape", s)
>>> shape("myshape")
```

> **Note**
>
> La classe `Shape` est utilisée en interne par la méthode `register_shape()` de différentes façons. Le développeur n'interagit avec la classe Shape *que* lorsqu'il utilise des formes composées comme montré ci-dessus !

25.1.7 Méthodes de TurtleScreen/Screen et leurs fonctions correspondantes

La plupart des exemples dans cette section font référence à une instance de TurtleScreen appelée `screen`.

Réglage de la fenêtre

`turtle.`**`bgcolor`**(**args*)

Paramètres

`args` -- chaîne spécifiant une couleur ou trois nombres dans l'intervalle *0..colormode* ou *n*-uplet de ces trois nombres

Définit ou renvoie la couleur de fond de l'écran de la tortue (*TurtleScreen* en anglais).

```
>>> screen.bgcolor("orange")
>>> screen.bgcolor()
'orange'
>>> screen.bgcolor("#800080")
>>> screen.bgcolor()
(128.0, 0.0, 128.0)
```

`turtle.`**`bgpic`**(*picname=None*)

Paramètres

`picname` -- a string, name of an image file (PNG, GIF, PGM, and PPM) or `"nopic"`, or `None`

Définit l'image de fond ou renvoie l'image de fond actuelle. Si *picname* est un nom de fichier, cette image et mis en image de fond. Si *picname* est `"nopic"`, l'image de fond sera supprimée si présente. SI *picname* est `None`, le nom du fichier de l'image de fond actuelle est renvoyé.

```
>>> screen.bgpic()
'nopic'
>>> screen.bgpic("landscape.gif")
>>> screen.bgpic()
"landscape.gif"
```

`turtle.`**`clear`**()

> **Note**
>
> Cette méthode TurtleScreen est disponible en tant que fonction globale seulement sous le nom `clearscreen`. La fonction globale `clear` est une fonction différente dérivée de la méthode Turtle `clear`.

`turtle.clearscreen()`

Supprime tous les dessins et toutes les tortues du TurtleScreen. Réinitialise le TurtleScreen maintenant vide à son état initial : fond blanc, pas d'image de fond, pas d'événement liés, et traçage activé.

`turtle.reset()`

> **Note**
>
> Cette méthode TurtleScreen est disponible en tant que fonction globale seulement sous le nom `resetscreen`. La fonction globale `reset` est une fonction différente dérivée de la méthode Turtle `reset`.

`turtle.resetscreen()`

Remet toutes les tortues à l'écran dans leur état initial.

`turtle.screensize(canvwidth=None, canvheight=None, bg=None)`

Paramètres

- **canvwidth** -- nombre entier positif, nouvelle largeur du canevas (zone sur laquelle se déplace la tortue), en pixels
- **canvheight** -- nombre entier positif, nouvelle hauteur du canevas, en pixels
- **bg** -- chaîne de caractères indiquant la couleur ou triplet de couleurs, nouvelle couleur de fond

Si aucun arguments ne sont passés, renvoie l'actuel *(canvaswidth, canvasheight)*. Sinon, redimensionne le canevas sur lequel les tortues dessinent. Ne modifiez pas la fenêtre de dessin. Pour observer les parties cachées du canevas, utilisez les barres de défilement. Avec cette méthode, on peut rendre visible les parties d'un dessin qui étaient en dehors du canevas précédemment.

```
>>> screen.screensize()
(400, 300)
>>> screen.screensize(2000,1500)
>>> screen.screensize()
(2000, 1500)
```

par exemple, chercher une tortue échappée de manière erronée

`turtle.setworldcoordinates(llx, lly, urx, ury)`

Paramètres

- **llx** -- un nombre, coordonnée x du coin inférieur gauche du canevas
- **lly** -- un nombre, la coordonnée y du coin inférieur gauche du canevas
- **urx** -- un nombre, la coordonnée x du coin supérieur droit du canevas
- **ury** -- un nombre, la coordonnée y du coin supérieur droit du canevas

Configure un système de coordonnées défini par l'utilisateur et bascule vers le mode "world" si nécessaire. Cela effectuera un `screen.reset()`. Si le mode "world" est déjà actif, tous les dessins sont re-déssinés par rapport aux nouveaux coordonnées.

ATTENTION : dans les systèmes de coordonnées définis par l'utilisateur, les angles peuvent apparaître déformés.

```
>>> screen.reset()
>>> screen.setworldcoordinates(-50,-7.5,50,7.5)
>>> for _ in range(72):
...     left(10)
...
>>> for _ in range(8):
...     left(45); fd(2)   # a regular octagon
```

Réglage de l'animation

`turtle.no_animation()`

Temporarily disable turtle animation. The code written inside the `no_animation` block will not be animated ; once the code block is exited, the drawing will appear.

```
>>> with screen.no_animation():
...     for dist in range(2, 400, 2):
...         fd(dist)
...         rt(90)
```

Ajouté dans la version 3.14.

`turtle.delay(delay=None)`

Paramètres

delay -- entier positif

Définit ou renvoie le délai (*delay*) de dessin en millisecondes. (Cet approximativement le temps passé entre deux mises à jour du canevas.) Plus le délai est long, plus l'animation sera lente.

Argument facultatif :

```
>>> screen.delay()
10
>>> screen.delay(5)
>>> screen.delay()
5
```

`turtle.tracer(n=None, delay=None)`

Paramètres

— **n** -- entier non-négatif
— **delay** -- entier non-négatif

Active/désactive les animations des tortues et définit le délai pour mettre à jour les dessins.Si *n* est passé, seulement les n-ièmes mises à jours régulières de l'écran seront vraiment effectuées. (Peut être utilisé pour accélérer le dessin de graphiques complexes.) Lorsqu'appelé sans arguments, renvoie la valeur actuelle de *n*. Le deuxième argument définit la valeur du délai (voir `delay()`).

```
>>> screen.tracer(8, 25)
>>> dist = 2
>>> for i in range(200):
...     fd(dist)
...     rt(90)
...     dist += 2
```

`turtle.update()`

Effectue une mise à jour de *TurtleScreen*. À utiliser lorsque le traceur est désactivé.

Voir aussi la méthode `speed()` de *RawTurtle*/*Turtle*.

Utilisation des événements concernant l'écran

`turtle.listen(xdummy=None, ydummy=None)`

Donne le focus à *TurtleScreen* (afin de collecter les événements clés). Des arguments factices sont fournis afin de pouvoir passer `listen()` à la méthode *onclick*.

`turtle.onkey(fun, key)`

`turtle.onkeyrelease(fun, key)`

Paramètres

— **fun** -- une fonction sans arguments ou `None`

— **key** -- une chaîne : clé (par exemple "*a*") ou clé symbole (Par exemple "*space*")

Lie *fun* à l'événement d'un relâchement d'une touche. Si *fun* est `None`, les événements liés sont supprimés. Remarque : Pour pouvoir enregistrer les événements lié au touches, TurtleScreen doit avoir le *focus* (fenêtre en premier plan). (Voir la méthode `listen()`.)

```
>>> def f():
...     fd(50)
...     lt(60)
...
>>> screen.onkey(f, "Up")
>>> screen.listen()
```

`turtle.`**`onkeypress`**(*fun*, *key=None*)

Paramètres

— **fun** -- une fonction sans arguments ou `None`
— **key** -- une chaîne : clé (par exemple "*a*") ou clé symbole (Par exemple "*space*")

Lie *fun* à l'événement d'un pressement de touche si *key* (touche) est donné, ou n'importe quelle touche si aucune touche n'est passée. Remarque : Pour pouvoir enregistrer des événements liés au touches, TurtleScreen doit être en premier plan. (voir la méthode `listen()`.)

```
>>> def f():
...     fd(50)
...
>>> screen.onkey(f, "Up")
>>> screen.listen()
```

`turtle.`**`onclick`**(*fun*, *btn=1*, *add=None*)

`turtle.`**`onscreenclick`**(*fun*, *btn=1*, *add=None*)

Paramètres

— **fun** -- une fonction à deux arguments qui sera appelée avec les coordonnées du point cliqué sur le canevas
— **btn** -- numéro du bouton de la souris, par défaut 1 (bouton de gauche)
— **add** -- `True` ou `False` — si `True`, un nouveau lien est ajouté, sinon il remplace un ancien lien

Crée un lien vers *fun* pour les événements de clique de la souris sur cet écran. Si *fun* est `None`, les liens existants sont supprimés.

Exemple pour une instance de TurtleScreen nommée `screen` et une instance Turtle nommée `turtle` :

```
>>> screen.onclick(turtle.goto) # Subsequently clicking into the TurtleScreen␣
↪will
>>>                             # make the turtle move to the clicked point.
>>> screen.onclick(None)        # remove event binding again
```

Note

Cette méthode de TurtleScreen est disponible en tant que fonction globale seulement sous le nom de `onscreenclick`. La fonction globale `onclick` est une autre fonction dérivée de la méthode Turtle `onclick`.

`turtle.`**`ontimer`**(*fun*, *t=0*)

Paramètres

— **fun** -- une fonction sans arguments
— **t** -- un nombre supérieur ou égal à 0

Installe un minuteur qui appelle *fun* après *t* millisecondes.

```
>>> running = True
>>> def f():
...     if running:
...         fd(50)
...         lt(60)
...         screen.ontimer(f, 250)
>>> f()   ### makes the turtle march around
>>> running = False
```

turtle.**mainloop**()

turtle.**done**()

Démarre la boucle d'événements - appelle la boucle principale de Tkinter. Doit être la dernière opération dan un programme graphique *turtle*. **Ne dois pas** être utilisé si un script est lancé depuis IDLE avec le mode `-n` (pas de sous processus) - pour une utilisation interactive des graphiques *turtle*

```
>>> screen.mainloop()
```

Méthodes de saisie

turtle.**textinput**(*title*, *prompt*)

Paramètres

— **title** -- chaîne de caractères
— **prompt** -- chaîne de caractères

Fait apparaitre une fenêtre pour entrer une chaine de caractères. Le paramètre *title* est le titre de la fenêtre, *prompt* est le texte expliquant quelle information écrire. Renvoie l'entrée utilisateur sous forme de chaîne. Si le dialogue est annulé, renvoie `None`.

```
>>> screen.textinput("NIM", "Name of first player:")
```

turtle.**numinput**(*title*, *prompt*, *default=None*, *minval=None*, *maxval=None*)

Paramètres

— **title** -- chaîne de caractères
— **prompt** -- chaîne de caractères
— **default** -- un nombre (facultatif)
— **minval** -- un nombre (facultatif)
— **maxval** -- un nombre (facultatif)

Pop up a dialog window for input of a number. title is the title of the dialog window, prompt is a text mostly describing what numerical information to input. default : default value, minval : minimum value for input, maxval : maximum value for input. The number input must be in the range minval .. maxval if these are given. If not, a hint is issued and the dialog remains open for correction. Return the number input. If the dialog is canceled, return `None`.

```
>>> screen.numinput("Poker", "Your stakes:", 1000, minval=10, maxval=10000)
```

Paramétrages et méthodes spéciales

turtle.**mode**(*mode=None*)

Paramètres

mode -- l'une des chaînes de caractères : `"standard"`, `"logo"` ou `"world"`

Règle le mode de la tortue ("*standard*", "*logo*" ou "*world*") et la réinitialise. Si le mode n'est pas donné, le mode actuel est renvoyé.

Le mode "*standard*" est compatible avec l'ancien `turtle`. Le mode "*logo*" est compatible avec la plupart des graphiques *turtle* Logo. Le mode "*world*" utilise des "coordonnées monde" (*world coordinates*) définis par l'utilisateur. **Attention** : Dans ce mode, les angles apparaissent déformés si le ratio unitaire de `x/y` n'est pas 1.

Mode	Orientation initiale de la tortue	angles positifs
"standard"	vers la droite (vers l'Est)	dans le sens inverse des aiguilles d'une montre
"logo"	vers le haut (vers le Nord)	dans le sens des aiguilles d'une montre

```
>>> mode("logo")    # resets turtle heading to north
>>> mode()
'logo'
```

turtle.**colormode**(*cmode=None*)

Paramètres

cmode -- l'une des valeurs suivantes : 1.0 ou 255

Renvoie le mode de couleur (*colormode*) ou le définit à 1.0 ou 255. Les valeurs *r*, *g* et *b* doivent aussi être dans la gamme `0..`*cmode*.

```
>>> screen.colormode(1)
>>> turtle.pencolor(240, 160, 80)
Traceback (most recent call last):
     ...
TurtleGraphicsError: bad color sequence: (240, 160, 80)
>>> screen.colormode()
1.0
>>> screen.colormode(255)
>>> screen.colormode()
255
>>> turtle.pencolor(240,160,80)
```

turtle.**getcanvas**()

Renvoie le canevas de ce TurtleScreen. Utile pour les initiés qui savent quoi faire avec un canevas Tkinter.

```
>>> cv = screen.getcanvas()
>>> cv
<turtle.ScrolledCanvas object ...>
```

turtle.**getshapes**()

Renvoie une liste de noms de toutes les formes actuellement disponibles pour les tortues.

```
>>> screen.getshapes()
['arrow', 'blank', 'circle', ..., 'turtle']
```

turtle.**register_shape**(*name*, *shape=None*)

turtle.**addshape**(*name*, *shape=None*)

There are four different ways to call this function :

(1) *name* is the name of an image file (PNG, GIF, PGM, and PPM) and *shape* is `None` : Install the corresponding image shape.

```
>>> screen.register_shape("turtle.gif")
```

> **Note**
>
> Les formes d'images *ne tournent pas* lorsque la tortue tourne, donc elles n'indiquent pas le cap de la tortue !

(2) *name* is an arbitrary string and *shape* is the name of an image file (PNG, GIF, PGM, and PPM) : Install the corresponding image shape.

```
>>> screen.register_shape("turtle", "turtle.gif")
```

> **Note**
>
> Les formes d'images *ne tournent pas* lorsque la tortue tourne, donc elles n'indiquent pas le cap de la tortue !

(3) *name* est une chaîne de caractères arbitraire et *shape* est un *n*-uplet de paires de coordonnées : Installe le polygone correspondant.

```
>>> screen.register_shape("triangle", ((5,-3), (0,5), (-5,-3)))
```

(4) *name* is an arbitrary string and *shape* is a (compound) `Shape` object : Install the corresponding compound shape.

Ajoute une forme de tortue a la liste des formes du TurtleScreen. Seulement les formes enregistrées de cette façon peuvent être utilisée avec la commande `shape(shapename)`.

Modifié dans la version 3.14 : Added support for PNG, PGM, and PPM image formats. Both a shape name and an image file name can be specified.

`turtle.`**`turtles`**`()`

Renvoie la liste des tortues présentes sur l'écran.

```
>>> for turtle in screen.turtles():
...     turtle.color("red")
```

`turtle.`**`window_height`**`()`

Renvoie la hauteur de la fenêtre de la tortue.

```
>>> screen.window_height()
480
```

`turtle.`**`window_width`**`()`

Renvoie la largeur de la fenêtre de la tortue.

```
>>> screen.window_width()
640
```

Méthodes spécifiques à Screen, non héritées de TurtleScreen

`turtle.`**`bye`**`()`

Éteins la fenêtre *turtlegraphics*.

`turtle.`**`exitonclick`**`()`

Lie la méthode `bye()` à un clique de souris sur l'écran (*Screen*).

Si la valeur de "*using_IDLE*" dans le dictionnaire de configuration est `False` (valeur par défaut), démarre aussi la boucle principale. Remarque : Si IDLE est lancé avec l'option `-n` (Pas de sous processus), Cette valeur devrait être définie à `True` dans `turtle.cfg`. Dans ce cas, la boucle principale d'IDLE est active aussi pour le script du client.

`turtle.`**`save`**`(`*filename*, *overwrite=False*`)`

Save the current turtle drawing (and turtles) as a PostScript file.

Paramètres

- **`filename`** -- the path of the saved PostScript file
- **`overwrite`** -- if `False` and there already exists a file with the given filename, then the function will raise a `FileExistsError`. If it is `True`, the file will be overwritten.

```
>>> screen.save("my_drawing.ps")
>>> screen.save("my_drawing.ps", overwrite=True)
```

Ajouté dans la version 3.14.

turtle.**setup**(*width=_CFG['width'], height=_CFG['height'], startx=_CFG['leftright'], starty=_CFG['topbottom']*)

Définit la taille et la position de la fenêtre principale. Les valeurs par défaut des arguments sont stockées dans le dictionnaire de configuration et peuvent être modifiées via un fichier `turtle.cfg`.

Paramètres

- **width** -- s'il s'agit d'un nombre entier, une taille en pixels, s'il s'agit d'un nombre flottant, une fraction de l'écran ; la valeur par défaut est de 50 % de l'écran
- **height** -- s'il s'agit d'un nombre entier, la hauteur en pixels, s'il s'agit d'un nombre flottant, une fraction de l'écran ; la valeur par défaut est 75 % de l'écran
- **startx** -- s'il s'agit d'un nombre positif, position de départ en pixels à partir du bord gauche de l'écran, s'il s'agit d'un nombre négatif, position de départ en pixels à partir du bord droit, si c'est `None`, centre la fenêtre horizontalement
- **starty** -- si positif, la position de départ en pixels depuis le haut de l'écran. Si négatif, depuis de bas de l'écran. Si `None`, Le centre de la fenêtre verticalement

```
>>> screen.setup (width=200, height=200, startx=0, starty=0)
>>>              # sets window to 200x200 pixels, in upper left of screen
>>> screen.setup(width=.75, height=0.5, startx=None, starty=None)
>>>              # sets window to 75% of screen by 50% of screen and centers
```

turtle.**title**(*titlestring*)

Paramètres

titlestring -- chaîne de caractères affichée dans la barre de titre de la fenêtre graphique de la tortue

Définit le titre de la fenêtre de la tortue comme *titlestring*.

```
>>> screen.title("Welcome to the turtle zoo!")
```

25.1.8 Classes publiques

class turtle.**RawTurtle**(*canvas*)

class turtle.**RawPen**(*canvas*)

Paramètres

canvas -- a `tkinter.Canvas`, a *`ScrolledCanvas`* or a *`TurtleScreen`*

Crée une tortue. Cette tortue à toutes les méthodes décrites ci-dessus comme "Méthode de Turtle/RawTurtle".

class turtle.**Turtle**

Sous-classe de RawTurtle, à la même interface mais dessine sur un objet `screen` par défaut créé automatiquement lorsque nécessaire pour la première fois.

class turtle.**TurtleScreen**(*cv*)

Paramètres

cv -- a `tkinter.Canvas`

Provides screen oriented methods like *`bgcolor()`* etc. that are described above.

class turtle.**Screen**

Sous-classess de TurtleScreen, avec *quatre nouvelles méthodes*.

class turtle.**ScrolledCanvas**(*master*)

Paramètres
master -- certain modules Tkinter pour contenir le ScrolledCanvas, c'est à dire, un canevas Tkinter avec des barres de défilement ajoutées

Utilisé par la classe Screen, qui fournit donc automatiquement un ScrolledCanvas comme terrain de jeu pour les tortues.

class turtle.**Shape**(*type_*, *data*)

Paramètres
type_ -- l'une des chaînes suivantes : `"polygon"`, `"image"` ou `"compound"`

Formes de modélisation de la structure des données. La paire `(type_, data)` doit suivre cette spécification :

*type_*	*données*
"polygon"	un polygone *n*-uplet, c'est-à-dire un *n*-uplet constitué de paires (chaque paire définissant des coordonnées)
"image"	une image (utilisée uniquement en interne sous ce format !)
"compound"	`None` (une forme composée doit être construite en utilisant la méthode `addcomponent()`)

addcomponent(*poly*, *fill*, *outline=None*)

Paramètres
— **poly** -- un polygone, c.-à-d. un *n*-uplet de paires de nombres
— **fill** -- une couleur de remplissage pour *poly*
— **outline** -- une couleur pour le contour du polygone (si elle est donnée)

Exemple :

```
>>> poly = ((0,0),(10,-5),(0,10),(-10,-5))
>>> s = Shape("compound")
>>> s.addcomponent(poly, "red", "blue")
>>> # ... add more components and then use register_shape()
```

Voir *Formes composées*.

class turtle.**Vec2D**(*x*, *y*)

Une classe de vecteur bidimensionnel, utilisée en tant que classe auxiliaire pour implémenter les graphiques *turtle*. Peut être utile pour les programmes graphiques faits avec *turtle*. Dérivé des *n*-uplets, donc un vecteur est un *n*-uplet !

Permet (pour les vecteurs *a*, *b* et le nombre *k*) :
— `a + b` addition de vecteurs
— `a - b` soustraction de deux vecteurs
— `a * b` produit scalaire
— `k * a` et `a * k` multiplication avec un scalaire
— `abs(a)` valeur absolue de a
— `a.rotate(angle)` rotation

25.1.9 Explication

A turtle object draws on a screen object, and there a number of key classes in the turtle object-oriented interface that can be used to create them and relate them to each other.

A `Turtle` instance will automatically create a `Screen` instance if one is not already present.

`Turtle` is a subclass of `RawTurtle`, which *doesn't* automatically create a drawing surface - a *canvas* will need to be provided or created for it. The *canvas* can be a `tkinter.Canvas`, `ScrolledCanvas` or `TurtleScreen`.

TurtleScreen is the basic drawing surface for a turtle. *Screen* is a subclass of TurtleScreen, and includes *some additional methods* for managing its appearance (including size and title) and behaviour. TurtleScreen's constructor needs a tkinter.Canvas or a *ScrolledCanvas* as an argument.

The functional interface for turtle graphics uses the various methods of Turtle and TurtleScreen/Screen. Behind the scenes, a screen object is automatically created whenever a function derived from a Screen method is called. Similarly, a turtle object is automatically created whenever any of the functions derived from a Turtle method is called.

To use multiple turtles on a screen, the object-oriented interface must be used.

25.1.10 Aide et configuration

Utilisation de l'aide

Les méthodes publiques des classes *Screen* et *Turtle* sont largement documentées dans les *docstrings*. Elles peuvent donc être utilisées comme aide en ligne via les fonctions d'aide de Python :

— Lors de l'utilisation d'IDLE, des info-bulles apparaissent avec la signature et les premières lignes de *docstring* de la fonction/méthode appelée.

— L'appel de *help()* sur les méthodes ou fonctions affichera les *docstrings* :

```
>>> help(Screen.bgcolor)
Help on method bgcolor in module turtle:

bgcolor(self, *args) unbound turtle.Screen method
    Set or return backgroundcolor of the TurtleScreen.

    Arguments (if given): a color string or three numbers
    in the range 0..colormode or a 3-tuple of such numbers.


    >>> screen.bgcolor("orange")
    >>> screen.bgcolor()
    "orange"
    >>> screen.bgcolor(0.5,0,0.5)
    >>> screen.bgcolor()
    "#800080"

>>> help(Turtle.penup)
Help on method penup in module turtle:

penup(self) unbound turtle.Turtle method
    Pull the pen up -- no drawing when moving.

    Aliases: penup | pu | up

    No argument

    >>> turtle.penup()
```

— Les *docstrings* des fonctions qui sont dérivées des méthodes ont une forme modifiée :

```
>>> help(bgcolor)
Help on function bgcolor in module turtle:

bgcolor(*args)
    Set or return backgroundcolor of the TurtleScreen.

    Arguments (if given): a color string or three numbers
    in the range 0..colormode or a 3-tuple of such numbers.
```

(suite sur la page suivante)

(suite de la page précédente)

```
    Example::

      >>> bgcolor("orange")
      >>> bgcolor()
      "orange"
      >>> bgcolor(0.5,0,0.5)
      >>> bgcolor()
      "#800080"

>>> help(penup)
Help on function penup in module turtle:

penup()
    Pull the pen up -- no drawing when moving.

    Aliases: penup | pu | up

    No argument

    Example:
    >>> penup()
```

Ces chaînes de documents modifiées sont créées automatiquement avec les définitions de fonctions qui sont dérivées des méthodes au moment de l'importation.

Traduction de chaînes de documents en différentes langues

Il est utile de créer un dictionnaire dont les clés sont les noms des méthodes et les valeurs sont les *docstrings* de méthodes publiques des classes Screen et Turtle.

`turtle.`**`write_docstringdict`**(*filename='turtle_docstringdict'*)

Paramètres

`filename` -- une chaîne de caractères, utilisée en tant que nom de fichier

Crée et écrit un dictionnaire de *docstrings* dans un script Python avec le nom donné. Cette fonction doit être appelée explicitement (elle n'est pas utilisée par les classes graphiques de *turtle*). Ce dictionnaire de *doctrings* sera écrit dans le script Python `filename.py`. Il sert de modèle pour la traduction des *docstrings* dans différentes langues.

Si vous (ou vos étudiants) veulent utiliser `turtle` avec de l'aide en ligne dans votre langue natale, vous devez traduire les *docstrings* et sauvegarder les fichiers résultants en, par exemple, `turtle_docstringdict_german.py`.

Si vous avez une entrée appropriée dans votre fichier `turtle.cfg`, ce dictionnaire est lu au moment de l'importation et remplace la *docstrings* originale en anglais par cette entrée.

Au moment de l'écriture de cette documentation, il n'existe seulement que des *docstrings* en Allemand et Italien. (Merci de faire vos demandes à glingl@aon.at.)

Comment configurer *Screen* et *Turtle*

La configuration par défaut imite l'apparence et le comportement de l'ancien module *turtle* pour pouvoir maintenir la meilleure compatibilité avec celui-ci.

Si vous voulez utiliser une configuration différente qui reflète mieux les fonctionnalités de ce module ou qui correspond mieux à vos besoins, par exemple pour un cours, vous pouvez préparer un ficher de configuration `turtle.cfg` qui sera lu au moment de l'importation et qui modifiera la configuration en utilisant les paramètres du fichier.

The built in configuration would correspond to the following `turtle.cfg` :

```
width = 0.5
height = 0.75
leftright = None
topbottom = None
canvwidth = 400
canvheight = 300
mode = standard
colormode = 1.0
delay = 10
undobuffersize = 1000
shape = classic
pencolor = black
fillcolor = black
resizemode = noresize
visible = True
language = english
exampleturtle = turtle
examplescreen = screen
title = Python Turtle Graphics
using_IDLE = False
```

Brève explication des entrées sélectionnées :

- The first four lines correspond to the arguments of the `Screen.setup` method.
- Line 5 and 6 correspond to the arguments of the method `Screen.screensize`.
- *shape* peut être n'importe quelle forme native, par exemple *arrow*, *turtle* etc. Pour plus d'informations, essayez `help(shape)`.
- If you want to use no fill color (i.e. make the turtle transparent), you have to write `fillcolor = ""` (but all nonempty strings must not have quotes in the cfg file).
- Si vous voulez refléter l'état de la tortue, vous devez utiliser `resizemode = auto`.
- If you set e.g. `language = italian` the docstringdict `turtle_docstringdict_italian.py` will be loaded at import time (if present on the import path, e.g. in the same directory as `turtle`).
- Les entrées *exampleturtle* et *examplescreen* définissent les noms de ces objets tels qu'ils apparaissent dans les *docstrings*. La transformation des méthodes-*docstrings* vers fonction-*docstrings* supprimera ces noms des *docstrings*.
- *using_IDLE* : Set this to `True` if you regularly work with IDLE and its `-n` switch ("no subprocess"). This will prevent `exitonclick()` to enter the mainloop.

Il peut y avoir un `turtle.cfg` dans le dossier où se situe `turtle` et un autre dans le dossier de travail courant. Ce dernier prendra le dessus.

Le dossier `Lib/turtledemo` contient un fichier `turtle.cfg`. Vous pouvez le prendre comme exemple et voir ses effets lorsque vous lancez les démos (il est préférable de ne pas le faire depuis la visionneuse de démos).

25.1.11 `turtledemo` — Scripts de démonstration

Le paquet `turtledemo` inclut un ensemble de scripts de démonstration. Ces scripts peuvent être lancés et observés en utilisant la visionneuse de démos comme suit :

```
python -m turtledemo
```

Alternativement, vous pouvez lancer les scripts de démo individuellement. Par exemple

```
python -m turtledemo.bytedesign
```

Le paquet `turtledemo` contient :

- Une visionneuse `__main__.py` qui peut être utilisée pour lire le code source de ces scripts et pour les faire tourner en même temps.
- Plusieurs script présentent les différentes fonctionnalités du module `turtle`. Les exemples peuvent être consultés via le menu *Examples*. Ils peuvent aussi être lancés de manière autonome.
- Un fichier exemple `turtle.cfg` montrant comment rédiger de tels fichiers.

Les scripts de démonstration sont :

Nom	Description	Caractéristiques
bytedesign	motif complexe de la tortue graphique classique	`tracer()`, temps mort, `update()`
chaos	graphiques dynamiques de Verhulst, cela démontre que les calculs de l'ordinateur peuvent générer des résultats qui vont parfois à l'encontre du bon sens	*world coordinates*
clock	horloge analogique indiquant l'heure de votre ordinateur	tortues sous forme des aiguilles d'horloge, sur minuterie
colormixer (mélangeur de couleurs)	des expériences en rouge, vert, bleu	`ondrag()`
forest (forêt)	3 arbres tracés par un parcours en largeur	*randomization* (répartition aléatoire)
fractalcurves	Courbes de Hilbert et de Koch	récursivité
lindenmayer	ethnomathématiques (kolams indiens)	*L-Système*
minimal_hanoi	Tours de Hanoï	Des tortues rectangulaires à la place des disques (*shape*, *shapesize*)
nim	jouez au classique jeu de *nim* avec trois piles de bâtons contre l'ordinateur.	tortues en tant que bâtons de *nim*, géré par des événements (clavier et souris)
paint (peinture)	programme de dessin extra minimaliste	`onclick()`
peace (paix)	basique	tortue : apparence et animation
penrose	tuiles apériodiques avec cerfs-volants et fléchettes	`stamp()`
planet_and_moon (planète et lune)	simulation d'un système gravitationnel	formes composées, `Vec2D`
rosette	un motif issu de l'article de *wikipedia* sur la tortue graphique	`clone()`, `undo()`
round_dance	tortues dansantes tournant par paires en sens inverse	formes composées, clones de la forme (*shapesize*), rotation, *get_shapepoly*, *update*
sorting_animate	démonstration visuelle des différentes méthodes de classement	alignement simple, répartition aléatoire
tree (arbre)	un arbre (tracé) par un parcours en largeur (à l'aide de générateurs)	`clone()`
two_canvases (deux toiles)	design simple	tortues sur deux canevas
yinyang	un autre exemple élémentaire	`circle()`

Amusez-vous !

25.1.12 Modifications depuis Python 2.6

- The methods `Turtle.tracer`, `Turtle.window_width` and `Turtle.window_height` have been eliminated. Methods with these names and functionality are now available only as methods of `Screen`. The functions derived from these remain available. (In fact already in Python 2.6 these methods were merely duplications of the corresponding `TurtleScreen`/`Screen` methods.)
- The method `Turtle.fill()` has been eliminated. The behaviour of `begin_fill()` and `end_fill()` have changed slightly : now every filling process must be completed with an `end_fill()` call.
- A method `Turtle.filling` has been added. It returns a boolean value : `True` if a filling process is under way, `False` otherwise. This behaviour corresponds to a `fill()` call without arguments in Python 2.6.

25.1.13 Modifications depuis Python 3.0

— The `Turtle` methods `shearfactor()`, `shapetransform()` and `get_shapepoly()` have been added. Thus the full range of regular linear transforms is now available for transforming turtle shapes. `tiltangle()` has been enhanced in functionality : it now can be used to get or set the tilt angle.
— The `Screen` method `onkeypress()` has been added as a complement to `onkey()`. As the latter binds actions to the key release event, an alias : `onkeyrelease()` was also added for it.
— The method `Screen.mainloop` has been added, so there is no longer a need to use the standalone `mainloop()` function when working with `Screen` and `Turtle` objects.
— Two input methods have been added : `Screen.textinput` and `Screen.numinput`. These pop up input dialogs and return strings and numbers respectively.

25.2 `cmd` --- Support for line-oriented command interpreters

Code source : Lib/cmd.py

La `Cmd` fournit une base simple permettant d'écrire des interpréteurs en ligne de commande. Ceux-ci sont souvent utiles pour piloter des tests, pour des outils administratifs, et pour des prototypes destinés à être intégrés à une interface plus sophistiquée.

class `cmd.`**`Cmd`**(*completekey='tab', stdin=None, stdout=None*)

Une instance de `Cmd` ou d'une classe en héritant est un *framework* orienté ligne de commande. Il n'y a pas de bonne raison d'instancier `Cmd` directement. Elle est plutôt utile en tant que classe mère d'une classe-interprète que vous définirez afin d'hériter des méthodes de `Cmd` et d'encapsuler les opérations.

L'argument facultatif *completekey* est le nom `readline` d'une touche de complétion. Si *completekey* ne vaut pas `None` et que `readline` est disponible, la complétion de commandes est faite automatiquement.

The default, `'tab'`, is treated specially, so that it refers to the `Tab` key on every `readline.backend`. Specifically, if `readline.backend` is `editline`, `Cmd` will use `'^I'` instead of `'tab'`. Note that other values are not treated this way, and might only work with a specific backend.

Les arguments facultatifs *stdin* et *stdout* spécifient les objets-fichiers de lecture et d'écriture que l'instance de Cmd ou d'une classe fille utilisera comme entrée et sortie. Si ces arguments ne sont pas spécifiés, ils prendront comme valeur par défaut `sys.stdin` et `sys.stdout`.

Si vous souhaitez qu'un *stdin* donné soit utilisé, assurez-vous que l'attribut `use_rawinput` de l'instance vaille `False`, faute de quoi *stdin* sera ignoré.

Modifié dans la version 3.13 : `completekey='tab'` is replaced by `'^I'` for `editline`.

25.2.1 Objets Cmd

Une instance de `Cmd` possède les méthodes suivantes :

`Cmd.`**`cmdloop`**(*intro=None*)

Affiche une invite de commande de manière répétée, accepte une entrée, soustrait un préfixe initial de l'entrée reçue et envoie aux méthodes d'opération la partie restante de l'entrée reçue.

L'argument facultatif est une bannière ou chaîne de caractères d'introduction à afficher avant la première invite de commande (il redéfinit l'attribut de classe `intro`).

Si le module `readline` est chargé, l'entrée héritera automatiquement d'une édition d'historique similaire à **`bash`** (Par exemple, `Control-P` reviendra à la dernière commande, `Control-N` avancera à la suivante, `Control-F` déplace le curseur vers la droite, `Control-B` déplace le curseur vers la gauche, etc...).

Une caractère de fin de fichier est transmis via la chaîne de caractères `'EOF'`.

An interpreter instance will recognize a command name `foo` if and only if it has a method `do_foo()`. As a special case, a line beginning with the character `'?'` is dispatched to the method `do_help()`. As another special case, a line beginning with the character `'!'` is dispatched to the method `do_shell()` (if such a method is defined).

This method will return when the `postcmd()` method returns a true value. The *stop* argument to `postcmd()` is the return value from the command's corresponding `do_*()` method.

If completion is enabled, completing commands will be done automatically, and completing of commands args is done by calling `complete_foo()` with arguments *text*, *line*, *begidx*, and *endidx*. *text* is the string prefix we are attempting to match : all returned matches must begin with it. *line* is the current input line with leading whitespace removed, *begidx* and *endidx* are the beginning and ending indexes of the prefix text, which could be used to provide different completion depending upon which position the argument is in.

`Cmd.`**`do_help`**(*arg*)

All subclasses of `Cmd` inherit a predefined `do_help()`. This method, called with an argument `'bar'`, invokes the corresponding method `help_bar()`, and if that is not present, prints the docstring of `do_bar()`, if available. With no argument, `do_help()` lists all available help topics (that is, all commands with corresponding `help_*()` methods or commands that have docstrings), and also lists any undocumented commands.

`Cmd.`**`onecmd`**(*str*)

Interpret the argument as though it had been typed in response to the prompt. This may be overridden, but should not normally need to be ; see the `precmd()` and `postcmd()` methods for useful execution hooks. The return value is a flag indicating whether interpretation of commands by the interpreter should stop. If there is a `do_*()` method for the command *str*, the return value of that method is returned, otherwise the return value from the `default()` method is returned.

`Cmd.`**`emptyline`**()

Méthode appelée quand une ligne vide est entrée en réponse à l'invite de commande. Si cette méthode n'est pas surchargée, elle répète la dernière commande non-vide entrée.

`Cmd.`**`default`**(*line*)

Méthode appelée lorsque le préfixe de commande d'une ligne entrée n'est pas reconnu. Si cette méthode n'est pas surchargée, elle affiche un message d'erreur et s'arrête.

`Cmd.`**`completedefault`**(*text*, *line*, *begidx*, *endidx*)

Method called to complete an input line when no command-specific `complete_*()` method is available. By default, it returns an empty list.

`Cmd.`**`columnize`**(*list*, *displaywidth=80*)

Méthode appelée pour afficher une liste de chaînes de caractères sous la forme d'un ensemble compact de colonnes. Chaque colonne est de largeur minimale. Les colonnes sont séparées par deux espaces pour faciliter la lecture.

`Cmd.`**`precmd`**(*line*)

Méthode de rappel exécutée juste avant que la ligne de commande *line* ne soit interprétée, mais après que l'invite de commande ait été généré et affiché. Cette méthode existe afin d'être surchargée par des classes filles de `Cmd`. La valeur de retour est utilisée comme la commande qui sera exécutée par la méthode `onecmd()`. L'implémentation de `precmd()` peut réécrire la commande ou simplement renvoyer *line* sans modification.

`Cmd.`**`postcmd`**(*stop*, *line*)

Méthode de rappel exécutée juste après qu'une commande ait été exécutée. Cette méthode existe afin d'être surchargée par des classes filles de `Cmd`. *line* est la ligne de commande ayant été exécutée et *stop* est un *flag* indiquant si l'exécution sera terminée après un appel à `postcmd()`. *stop* sera la valeur de retour de `onecmd()`. La valeur de retour de cette méthode sera utilisée comme nouvelle valeur pour le *flag* interne correspondant à *stop*. Renvoyer *False* permettra à l'interprétation de continuer.

`Cmd.`**`preloop`**()

Méthode de rappel exécutée une fois lorsque `cmdloop()` est appelée. Cette méthode existe afin d'être surchargée par des classes filles de `Cmd`.

`Cmd.`**`postloop`**()

Méthode de rappel exécutée une fois lorsque `cmdloop()` va s'arrêter. Cette méthode existe afin d'être surchargée par des classes filles de `Cmd`.

Les instances de classes filles de `Cmd` possèdent des variables d'instance publiques :

Cmd.**prompt**

L'invite de commande affiché pour solliciter une entrée.

Cmd.**identchars**

La chaîne de caractères acceptée en tant que préfixe de commande.

Cmd.**lastcmd**

Le dernier préfixe de commande non-vide vu.

Cmd.**cmdqueue**

Une liste de lignes entrées en file. La liste *cmdqueue* est vérifiée dans `cmdloop()` lorsqu'une nouvelle entrée est nécessitée ; si elle n'est pas vide, ses éléments seront traités dans l'ordre, comme si ils avaient entrés dans l'invite de commande.

Cmd.**intro**

Une chaîne de caractères à afficher en introduction ou bannière. Peut être surchargée en passant un argument à la méthode `cmdloop()`.

Cmd.**doc_header**

L'en-tête à afficher si la sortie de l'aide possède une section pour les commandes documentées.

Cmd.**misc_header**

The header to issue if the help output has a section for miscellaneous help topics (that is, there are `help_*()` methods without corresponding `do_*()` methods).

Cmd.**undoc_header**

The header to issue if the help output has a section for undocumented commands (that is, there are `do_*()` methods without corresponding `help_*()` methods).

Cmd.**ruler**

Le caractère utilisé pour afficher des lignes de séparation sous les en-têtes de messages d'aide. Si il est vide, aucune ligne de séparation n'est affichée. Par défaut, ce caractère vaut `'='`.

Cmd.**use_rawinput**

A flag, defaulting to true. If true, `cmdloop()` uses `input()` to display a prompt and read the next command ; if false, `sys.stdout.write()` and `sys.stdin.readline()` are used. (This means that by importing `readline`, on systems that support it, the interpreter will automatically support **Emacs**-like line editing and command-history keystrokes.)

25.2.2 Exemple

Le module `cmd` est utile pour produire des invites de commande permettant à l'utilisateur de travailler avec un programme de manière interactive.

Cette section présente un exemple simple de comment produire une invite de commande autour de quelques commandes du module `turtle`.

Basic turtle commands such as `forward()` are added to a `Cmd` subclass with method named `do_forward()`. The argument is converted to a number and dispatched to the turtle module. The docstring is used in the help utility provided by the shell.

The example also includes a basic record and playback facility implemented with the `precmd()` method which is responsible for converting the input to lowercase and writing the commands to a file. The `do_playback()` method reads the file and adds the recorded commands to the `cmdqueue` for immediate playback :

```
import cmd, sys
from turtle import *

class TurtleShell(cmd.Cmd):
    intro = 'Welcome to the turtle shell.   Type help or ? to list commands.\n'
    prompt = '(turtle) '
```

(suite sur la page suivante)

(suite de la page précédente)

```
    file = None

    # ----- basic turtle commands -----
    def do_forward(self, arg):
        'Move the turtle forward by the specified distance:  FORWARD 10'
        forward(*parse(arg))
    def do_right(self, arg):
        'Turn turtle right by given number of degrees:  RIGHT 20'
        right(*parse(arg))
    def do_left(self, arg):
        'Turn turtle left by given number of degrees:  LEFT 90'
        left(*parse(arg))
    def do_goto(self, arg):
        'Move turtle to an absolute position with changing orientation.  GOTO 100␣
↪200'
        goto(*parse(arg))
    def do_home(self, arg):
        'Return turtle to the home position:  HOME'
        home()
    def do_circle(self, arg):
        'Draw circle with given radius an options extent and steps:  CIRCLE 50'
        circle(*parse(arg))
    def do_position(self, arg):
        'Print the current turtle position:  POSITION'
        print('Current position is %d %d\n' % position())
    def do_heading(self, arg):
        'Print the current turtle heading in degrees:  HEADING'
        print('Current heading is %d\n' % (heading(),))
    def do_color(self, arg):
        'Set the color:  COLOR BLUE'
        color(arg.lower())
    def do_undo(self, arg):
        'Undo (repeatedly) the last turtle action(s):  UNDO'
    def do_reset(self, arg):
        'Clear the screen and return turtle to center:  RESET'
        reset()
    def do_bye(self, arg):
        'Stop recording, close the turtle window, and exit:  BYE'
        print('Thank you for using Turtle')
        self.close()
        bye()
        return True

    # ----- record and playback -----
    def do_record(self, arg):
        'Save future commands to filename:  RECORD rose.cmd'
        self.file = open(arg, 'w')
    def do_playback(self, arg):
        'Playback commands from a file:  PLAYBACK rose.cmd'
        self.close()
        with open(arg) as f:
            self.cmdqueue.extend(f.read().splitlines())
    def precmd(self, line):
        line = line.lower()
        if self.file and 'playback' not in line:
            print(line, file=self.file)
```

(suite sur la page suivante)

(suite de la page précédente)

```
        return line
    def close(self):
        if self.file:
            self.file.close()
            self.file = None

def parse(arg):
    'Convert a series of zero or more numbers to an argument tuple'
    return tuple(map(int, arg.split()))

if __name__ == '__main__':
    TurtleShell().cmdloop()
```

Voici une session d'exemple avec l'invite de commande *turtle*. Elle montre les fonctions d'aide, utilise les lignes vides pour répéter des commandes et montre l'utilitaire de *playback* :

```
Welcome to the turtle shell.   Type help or ? to list commands.

(turtle) ?

Documented commands (type help <topic>):
========================================
bye     color    goto     home  playback  record  right
circle  forward  heading  left  position  reset   undo

(turtle) help forward
Move the turtle forward by the specified distance:  FORWARD 10
(turtle) record spiral.cmd
(turtle) position
Current position is 0 0

(turtle) heading
Current heading is 0

(turtle) reset
(turtle) circle 20
(turtle) right 30
(turtle) circle 40
(turtle) right 30
(turtle) circle 60
(turtle) right 30
(turtle) circle 80
(turtle) right 30
(turtle) circle 100
(turtle) right 30
(turtle) circle 120
(turtle) right 30
(turtle) circle 120
(turtle) heading
Current heading is 180

(turtle) forward 100
(turtle)
(turtle) right 90
(turtle) forward 100
(turtle)
```

(suite sur la page suivante)

(suite de la page précédente)

```
(turtle) right 90
(turtle) forward 400
(turtle) right 90
(turtle) forward 500
(turtle) right 90
(turtle) forward 400
(turtle) right 90
(turtle) forward 300
(turtle) playback spiral.cmd
Current position is 0 0

Current heading is 0

Current heading is 180

(turtle) bye
Thank you for using Turtle
```

25.3 `shlex` --- Simple lexical analysis

Code source : Lib/shlex.py

The `shlex` class makes it easy to write lexical analyzers for simple syntaxes resembling that of the Unix shell. This will often be useful for writing minilanguages, (for example, in run control files for Python applications) or for parsing quoted strings.

The `shlex` module defines the following functions :

`shlex.split`(*s*, *comments=False*, *posix=True*)

Split the string *s* using shell-like syntax. If *comments* is `False` (the default), the parsing of comments in the given string will be disabled (setting the `commenters` attribute of the `shlex` instance to the empty string). This function operates in POSIX mode by default, but uses non-POSIX mode if the *posix* argument is false.

Modifié dans la version 3.12 : Passing `None` for *s* argument now raises an exception, rather than reading `sys.stdin`.

`shlex.join`(*split_command*)

Concatenate the tokens of the list *split_command* and return a string. This function is the inverse of `split()`.

```
>>> from shlex import join
>>> print(join(['echo', '-n', 'Multiple words']))
echo -n 'Multiple words'
```

The returned value is shell-escaped to protect against injection vulnerabilities (see `quote()`).

Ajouté dans la version 3.8.

`shlex.quote`(*s*)

Return a shell-escaped version of the string *s*. The returned value is a string that can safely be used as one token in a shell command line, for cases where you cannot use a list.

> **Avertissement**
>
> The `shlex` module is **only designed for Unix shells**.

> The `quote()` function is not guaranteed to be correct on non-POSIX compliant shells or shells from other operating systems such as Windows. Executing commands quoted by this module on such shells can open up the possibility of a command injection vulnerability.
>
> Consider using functions that pass command arguments with lists such as `subprocess.run()` with `shell=False`.

This idiom would be unsafe :

```
>>> filename = 'somefile; rm -rf ~'
>>> command = 'ls -l {}'.format(filename)
>>> print(command)  # executed by a shell: boom!
ls -l somefile; rm -rf ~
```

`quote()` lets you plug the security hole :

```
>>> from shlex import quote
>>> command = 'ls -l {}'.format(quote(filename))
>>> print(command)
ls -l 'somefile; rm -rf ~'
>>> remote_command = 'ssh home {}'.format(quote(command))
>>> print(remote_command)
ssh home 'ls -l '"'"'somefile; rm -rf ~'"'"'
```

The quoting is compatible with UNIX shells and with `split()` :

```
>>> from shlex import split
>>> remote_command = split(remote_command)
>>> remote_command
['ssh', 'home', "ls -l 'somefile; rm -rf ~'"]
>>> command = split(remote_command[-1])
>>> command
['ls', '-l', 'somefile; rm -rf ~']
```

Ajouté dans la version 3.3.

The `shlex` module defines the following class :

class shlex.**shlex**(*instream=None*, *infile=None*, *posix=False*, *punctuation_chars=False*)

A `shlex` instance or subclass instance is a lexical analyzer object. The initialization argument, if present, specifies where to read characters from. It must be a file-/stream-like object with `read()` and `readline()` methods, or a string. If no argument is given, input will be taken from `sys.stdin`. The second optional argument is a filename string, which sets the initial value of the `infile` attribute. If the *instream* argument is omitted or equal to `sys.stdin`, this second argument defaults to "stdin". The *posix* argument defines the operational mode : when *posix* is not true (default), the `shlex` instance will operate in compatibility mode. When operating in POSIX mode, `shlex` will try to be as close as possible to the POSIX shell parsing rules. The *punctuation_chars* argument provides a way to make the behaviour even closer to how real shells parse. This can take a number of values : the default value, `False`, preserves the behaviour seen under Python 3.5 and earlier. If set to `True`, then parsing of the characters `();<>|&` is changed : any run of these characters (considered punctuation characters) is returned as a single token. If set to a non-empty string of characters, those characters will be used as the punctuation characters. Any characters in the `wordchars` attribute that appear in *punctuation_chars* will be removed from `wordchars`. See *Improved Compatibility with Shells* for more information. *punctuation_chars* can be set only upon `shlex` instance creation and can't be modified later.

Modifié dans la version 3.6 : The *punctuation_chars* parameter was added.

Voir aussi

Module *configparser*
Parser for configuration files similar to the Windows `.ini` files.

25.3.1 shlex Objects

A *shlex* instance has the following methods :

`shlex.get_token()`

Return a token. If tokens have been stacked using *push_token()*, pop a token off the stack. Otherwise, read one from the input stream. If reading encounters an immediate end-of-file, *eof* is returned (the empty string (`''`) in non-POSIX mode, and `None` in POSIX mode).

`shlex.push_token(str)`

Push the argument onto the token stack.

`shlex.read_token()`

Read a raw token. Ignore the pushback stack, and do not interpret source requests. (This is not ordinarily a useful entry point, and is documented here only for the sake of completeness.)

`shlex.sourcehook(filename)`

When *shlex* detects a source request (see *source* below) this method is given the following token as argument, and expected to return a tuple consisting of a filename and an open file-like object.

Normally, this method first strips any quotes off the argument. If the result is an absolute pathname, or there was no previous source request in effect, or the previous source was a stream (such as `sys.stdin`), the result is left alone. Otherwise, if the result is a relative pathname, the directory part of the name of the file immediately before it on the source inclusion stack is prepended (this behavior is like the way the C preprocessor handles `#include "file.h"`).

The result of the manipulations is treated as a filename, and returned as the first component of the tuple, with *open()* called on it to yield the second component. (Note : this is the reverse of the order of arguments in instance initialization !)

This hook is exposed so that you can use it to implement directory search paths, addition of file extensions, and other namespace hacks. There is no corresponding 'close' hook, but a shlex instance will call the *close()* method of the sourced input stream when it returns EOF.

For more explicit control of source stacking, use the *push_source()* and *pop_source()* methods.

`shlex.push_source(newstream, newfile=None)`

Push an input source stream onto the input stack. If the filename argument is specified it will later be available for use in error messages. This is the same method used internally by the *sourcehook()* method.

`shlex.pop_source()`

Pop the last-pushed input source from the input stack. This is the same method used internally when the lexer reaches EOF on a stacked input stream.

`shlex.error_leader(infile=None, lineno=None)`

This method generates an error message leader in the format of a Unix C compiler error label ; the format is `'"%s", line %d: '`, where the `%s` is replaced with the name of the current source file and the `%d` with the current input line number (the optional arguments can be used to override these).

This convenience is provided to encourage *shlex* users to generate error messages in the standard, parseable format understood by Emacs and other Unix tools.

Instances of *shlex* subclasses have some public instance variables which either control lexical analysis or can be used for debugging :

`shlex.commenters`

The string of characters that are recognized as comment beginners. All characters from the comment beginner to end of line are ignored. Includes just `'#'` by default.

`shlex.wordchars`

The string of characters that will accumulate into multi-character tokens. By default, includes all ASCII alphanumerics and underscore. In POSIX mode, the accented characters in the Latin-1 set are also included. If *punctuation_chars* is not empty, the characters `~-./*?=`, which can appear in filename specifications and command line parameters, will also be included in this attribute, and any characters which appear in `punctuation_chars` will be removed from `wordchars` if they are present there. If *whitespace_split* is set to `True`, this will have no effect.

`shlex.whitespace`

Characters that will be considered whitespace and skipped. Whitespace bounds tokens. By default, includes space, tab, linefeed and carriage-return.

`shlex.escape`

Characters that will be considered as escape. This will be only used in POSIX mode, and includes just `'\'` by default.

`shlex.quotes`

Characters that will be considered string quotes. The token accumulates until the same quote is encountered again (thus, different quote types protect each other as in the shell.) By default, includes ASCII single and double quotes.

`shlex.escapedquotes`

Characters in *quotes* that will interpret escape characters defined in *escape*. This is only used in POSIX mode, and includes just `'"'` by default.

`shlex.whitespace_split`

If `True`, tokens will only be split in whitespaces. This is useful, for example, for parsing command lines with *shlex*, getting tokens in a similar way to shell arguments. When used in combination with *punctuation_chars*, tokens will be split on whitespace in addition to those characters.

Modifié dans la version 3.8 : The *punctuation_chars* attribute was made compatible with the *whitespace_split* attribute.

`shlex.infile`

The name of the current input file, as initially set at class instantiation time or stacked by later source requests. It may be useful to examine this when constructing error messages.

`shlex.instream`

The input stream from which this *shlex* instance is reading characters.

`shlex.source`

This attribute is `None` by default. If you assign a string to it, that string will be recognized as a lexical-level inclusion request similar to the `source` keyword in various shells. That is, the immediately following token will be opened as a filename and input will be taken from that stream until EOF, at which point the *close()* method of that stream will be called and the input source will again become the original input stream. Source requests may be stacked any number of levels deep.

`shlex.debug`

If this attribute is numeric and `1` or more, a *shlex* instance will print verbose progress output on its behavior. If you need to use this, you can read the module source code to learn the details.

`shlex.lineno`

Source line number (count of newlines seen so far plus one).

`shlex.token`

The token buffer. It may be useful to examine this when catching exceptions.

`shlex.eof`

Token used to determine end of file. This will be set to the empty string (`''`), in non-POSIX mode, and to `None` in POSIX mode.

`shlex.punctuation_chars`

A read-only property. Characters that will be considered punctuation. Runs of punctuation characters will be returned as a single token. However, note that no semantic validity checking will be performed : for example, '»>' could be returned as a token, even though it may not be recognised as such by shells.

Ajouté dans la version 3.6.

25.3.2 Parsing Rules

When operating in non-POSIX mode, *shlex* will try to obey to the following rules.

- Quote characters are not recognized within words (`Do"Not"Separate` is parsed as the single word `Do"Not"Separate`);
- Escape characters are not recognized;
- Enclosing characters in quotes preserve the literal value of all characters within the quotes;
- Closing quotes separate words (`"Do"Separate` is parsed as `"Do"` and `Separate`);
- If *whitespace_split* is `False`, any character not declared to be a word character, whitespace, or a quote will be returned as a single-character token. If it is `True`, *shlex* will only split words in whitespaces;
- EOF is signaled with an empty string (`''`);
- It's not possible to parse empty strings, even if quoted.

When operating in POSIX mode, *shlex* will try to obey to the following parsing rules.

- Quotes are stripped out, and do not separate words (`"Do"Not"Separate"` is parsed as the single word `DoNotSeparate`);
- Non-quoted escape characters (e.g. `'\'`) preserve the literal value of the next character that follows;
- Enclosing characters in quotes which are not part of *escapedquotes* (e.g. `"'"`) preserve the literal value of all characters within the quotes;
- Enclosing characters in quotes which are part of *escapedquotes* (e.g. `'"'`) preserves the literal value of all characters within the quotes, with the exception of the characters mentioned in *escape*. The escape characters retain its special meaning only when followed by the quote in use, or the escape character itself. Otherwise the escape character will be considered a normal character.
- EOF is signaled with a *None* value;
- Quoted empty strings (`''`) are allowed.

25.3.3 Improved Compatibility with Shells

Ajouté dans la version 3.6.

The *shlex* class provides compatibility with the parsing performed by common Unix shells like `bash`, `dash`, and `sh`. To take advantage of this compatibility, specify the `punctuation_chars` argument in the constructor. This defaults to `False`, which preserves pre-3.6 behaviour. However, if it is set to `True`, then parsing of the characters `();<>|&` is changed : any run of these characters is returned as a single token. While this is short of a full parser for shells (which would be out of scope for the standard library, given the multiplicity of shells out there), it does allow you to perform processing of command lines more easily than you could otherwise. To illustrate, you can see the difference in the following snippet :

```
>>> import shlex
>>> text = "a && b; c && d || e; f >'abc'; (def \"ghi\")"
>>> s = shlex.shlex(text, posix=True)
>>> s.whitespace_split = True
>>> list(s)
['a', '&&', 'b;', 'c', '&&', 'd', '||', 'e;', 'f', '>abc;', '(def', 'ghi)']
>>> s = shlex.shlex(text, posix=True, punctuation_chars=True)
>>> s.whitespace_split = True
>>> list(s)
['a', '&&', 'b', ';', 'c', '&&', 'd', '||', 'e', ';', 'f', '>', 'abc', ';',
'(', 'def', 'ghi', ')']
```

Of course, tokens will be returned which are not valid for shells, and you'll need to implement your own error checks on the returned tokens.

Instead of passing `True` as the value for the punctuation_chars parameter, you can pass a string with specific characters, which will be used to determine which characters constitute punctuation. For example :

```
>>> import shlex
>>> s = shlex.shlex("a && b || c", punctuation_chars="|")
>>> list(s)
['a', '&', '&', 'b', '||', 'c']
```

Note

When `punctuation_chars` is specified, the *wordchars* attribute is augmented with the characters `~-./*?=`. That is because these characters can appear in file names (including wildcards) and command-line arguments (e.g. `--color=auto`). Hence :

```
>>> import shlex
>>> s = shlex.shlex('~/a && b-c --color=auto || d *.py?',
...                 punctuation_chars=True)
>>> list(s)
['~/a', '&&', 'b-c', '--color=auto', '||', 'd', '*.py?']
```

However, to match the shell as closely as possible, it is recommended to always use `posix` and *whitespace_split* when using *punctuation_chars*, which will negate *wordchars* entirely.

For best effect, `punctuation_chars` should be set in conjunction with `posix=True`. (Note that `posix=False` is the default for *shlex*.)

CHAPITRE 26

Interfaces Utilisateur Graphiques avec Tk

Tk/Tcl has long been an integral part of Python. It provides a robust and platform independent windowing toolkit, that is available to Python programmers using the `tkinter` package, and its extension, the `tkinter.ttk` module.

The `tkinter` package is a thin object-oriented layer on top of Tcl/Tk. To use `tkinter`, you don't need to write Tcl code, but you will need to consult the Tk documentation, and occasionally the Tcl documentation. `tkinter` is a set of wrappers that implement the Tk widgets as Python classes.

`tkinter`'s chief virtues are that it is fast, and that it usually comes bundled with Python. Although its standard documentation is weak, good material is available, which includes : references, tutorials, a book and others. `tkinter` is also famous for having an outdated look and feel, which has been vastly improved in Tk 8.5. Nevertheless, there are many other GUI libraries that you could be interested in. The Python wiki lists several alternative GUI frameworks and tools.

26.1 `tkinter` --- Python interface to Tcl/Tk

Code source : Lib/tkinter/__init__.py

Le paquet `tkinter` (« interface Tk ») est l'interface Python standard de la boîte à outils d'interface utilisateur graphique (GUI) *Tcl/Tk*. *Tk* et `tkinter` sont disponibles sur la plupart des plates-formes Unix, y compris macOS, ainsi que sur les systèmes Windows.

Exécuter `python -m tkinter` depuis la ligne de commande ouvre une fenêtre de démonstration d'une interface *Tk* simple, vous indiquant que `tkinter` est correctement installé sur votre système et indiquant également quelle version de *Tcl/Tk* est installée ; vous pouvez donc lire la documentation *Tcl/Tk* spécifique à cette version.

Tkinter prend en charge une gamme de versions *Tcl/Tk*, construites avec ou sans prise en charge des fils d'exécution multiples. La version binaire officielle de Python utilise *Tcl/Tk* 8.6 gérant les fils d'exécution multiples. Voir le code source du module `_tkinter` pour plus d'informations sur les versions prises en charge.

Tkinter n'est pas une simple enveloppe, mais ajoute une bonne partie de sa propre logique pour rendre l'expérience plus *pythonique*. Cette documentation se concentre sur ces ajouts et modifications, et se réfère à la documentation officielle de *Tcl/Tk* pour les détails qui restent inchangés.

Note

Tcl/Tk 8.5 (2007) a introduit un ensemble moderne de composants d'interface utilisateur thématiques ainsi qu'une nouvelle API pour les utiliser. Les anciennes et les nouvelles API sont toujours disponibles. La plupart des documents que vous trouverez en ligne utilisent toujours l'ancienne API et peuvent être terriblement obsolètes.

Voir aussi

— **TkDocs**
Tutoriel complet sur la création d'interfaces utilisateur avec *Tkinter*. Explique les concepts clés et illustre les approches recommandées à l'aide de l'API moderne.

— **Référence *Tkinter* 8.5 : une interface graphique pour Python**
Documentation de référence pour *Tkinter* 8.5 détaillant les classes, méthodes et options disponibles (ressource en anglais).

Ressources *Tcl*/Tk :

— **Commandes *Tk***
Référence complète de chacune des commandes *Tcl/Tk* sous-jacentes utilisées par *Tkinter* (ressource en anglais).

— **Page d'accueil de Tcl/Tk**
Documentation supplémentaire et liens vers le développement principal de *Tcl/Tk*.

Livres :

— **Modern Tkinter for Busy Python Developers**
par Mark Roseman. (ISBN 978-1999149567)

— **Python GUI programming with Tkinter**
By Alan D. Moore. (ISBN 978-1788835886)

— **Programming Python**
livre de Mark Lutz, qui couvre excellemment bien *Tkinter* (ISBN 978-0596158101).

— **Tcl and the Tk Toolkit (2nd edition)**
par John Ousterhout, inventeur de *Tcl/Tk*, et Ken Jones ; ne couvre pas *Tkinter*. (ISBN 978-0321336330)

26.1.1 Architecture

Tcl/Tk n'est pas une bibliothèque unique mais se compose plutôt de quelques modules distincts, chacun avec des fonctionnalités distinctes et sa propre documentation officielle. Les versions binaires de Python sont également livrées avec un module complémentaire.

Tcl
Tcl est un langage de programmation interprété dynamique, tout comme Python. Bien qu'il puisse être utilisé seul comme langage de programmation à usage général, il est le plus souvent intégré dans les applications C en tant que moteur de script ou interface avec la boîte à outils *Tk*. La bibliothèque *Tcl* a une interface C pour créer et gérer une ou plusieurs instances d'un interpréteur *Tcl*, exécuter des commandes et des scripts *Tcl* dans ces instances et ajouter des commandes personnalisées implémentées en *Tcl* ou C. Chaque interpréteur a une file d'attente d'événements et il y a facilités pour lui envoyer des événements et les traiter. Contrairement à Python, le modèle d'exécution de *Tcl* est conçu autour du multitâches coopératif, et *Tkinter* comble cette différence (voir *Modèle de thread* pour plus de détails).

Tk
Tk est un paquet *Tcl* implémenté en C qui ajoute des commandes personnalisées pour créer et manipuler des widgets d'interface graphique. Chaque objet `Tk` embarque sa propre instance d'interpréteur *Tcl* avec *Tk* chargé dedans. Les widgets de *Tk* sont très personnalisables, mais au prix d'une apparence datée. *Tk* utilise la file d'attente d'événements de *Tcl* pour générer et traiter les événements de l'interface graphique.

Ttk
Themed Tk (*Ttk*) est une nouvelle famille de widgets *Tk* qui offrent une bien meilleure apparence sur différentes plates-formes que la plupart des widgets *Tk* classiques. *Ttk* est distribué dans le cadre de *Tk*, à partir de la version 8.5 de *Tk*. Les liaisons Python sont fournies dans un module séparé, `tkinter.ttk`.

En interne, *Tk* et Ttk utilisent les fonctionnalités du système d'exploitation sous-jacent, c'est-à-dire Xlib sur Unix/X11, Cocoa sur macOS, GDI sur Windows.

Lorsque votre application Python utilise une classe dans *Tkinter*, par exemple pour créer un widget, le module `tkinter` assemble d'abord une chaîne de commande *Tcl/Tk*. Il passe cette chaîne de commande *Tcl* à un module binaire interne `_tkinter`, qui appelle ensuite l'interpréteur *Tcl* pour l'évaluer. L'interpréteur *Tcl* appelle alors les paquets *Tk* et/ou Ttk, qui à leur tour font des appels à Xlib, Cocoa ou GDI.

26.1.2 Modules *Tkinter*

La prise en charge de *Tkinter* est répartie sur plusieurs modules. La plupart des applications auront besoin du module principal `tkinter`, ainsi que du module `tkinter.ttk`, qui fournit l'ensemble de widgets thématiques modernes et l'API :

```
from tkinter import *
from tkinter import ttk
```

class `tkinter.`**`Tk`**(*screenName=None*, *baseName=None*, *className='Tk'*, *useTk=True*, *sync=False*, *use=None*)

Construit un widget *Tk* de niveau supérieur, qui est généralement la fenêtre principale d'une application, et initialise un interpréteur *Tcl* pour ce widget. Chaque instance a son propre interpréteur *Tcl* associé.

La classe `Tk` est généralement instanciée en utilisant toutes les valeurs par défaut. Cependant, les arguments nommés suivants sont actuellement reconnus :

screenName
Lorsqu'il est donné (sous forme de chaîne), définit la variable d'environnement `DISPLAY`. (X11 uniquement)

baseName
Nom du fichier de profil. Par défaut, *baseName* est dérivé du nom du programme (`sys.argv[0]`).

className
Nom de la classe de widget. Utilisé comme fichier de profil et aussi comme nom avec lequel *Tcl* est invoqué (*argv0* dans *interp*).

useTk
S'il vaut `True`, initialise le sous-système *Tk*. La fonction `tkinter.*Tcl*()` définit ceci sur `False`.

sync
Si `True`, exécute toutes les commandes du serveur X de manière synchrone, afin que les erreurs soient signalées immédiatement. Peut être utilisé pour le débogage. (X11 uniquement)

use
Spécifie l'*id* de la fenêtre dans laquelle intégrer l'application, au lieu de la créer en tant que fenêtre de niveau supérieur indépendante. *id* doit être spécifié de la même manière que la valeur de l'option -use pour les widgets de niveau supérieur (c'est-à-dire qu'il a une forme semblable à celle renvoyée par `winfo_id()`).

Notez que sur certaines plates-formes, cela ne fonctionne correctement que si *id* fait référence à un cadre *Tk* ou à un niveau supérieur dont l'option -container est activée.

`Tk` reads and interprets profile files, named `.className.tcl` and `.baseName.tcl`, into the Tcl interpreter and calls `exec()` on the contents of `.className.py` and `.baseName.py`. The path for the profile files is the `HOME` environment variable or, if that isn't defined, then `os.curdir`.

tk

L'objet d'application *Tk* créé en instanciant `Tk`. Cela donne accès à l'interpréteur *Tcl*. Chaque widget auquel est attachée la même instance de `Tk` a la même valeur pour son attribut `tk`.

master

L'objet widget qui contient ce widget. Pour `Tk`, le *master* est `None` car c'est la fenêtre principale. Les termes *master* et *parent* sont similaires et parfois utilisés de manière interchangeable comme noms d'arguments ; cependant, appeler `winfo_parent()` renvoie une chaîne du nom du widget alors que `master` renvoie l'objet. *parent/enfant* reflète la relation arborescente tandis que *maître/esclave* reflète la structure du conteneur.

children

Les descendants immédiats de ce widget en tant que `dict` avec les noms des widgets enfants comme clés et les objets d'instance enfants comme valeurs.

`tkinter.Tcl`(*screenName=None*, *baseName=None*, *className='Tk'*, *useTk=False*)

La fonction `Tcl()` est une fonction fabrique qui crée un objet similaire à celui créé par la classe `Tk`, sauf qu'elle n'initialise pas le sous-système *Tk*. Ceci est le plus souvent utile lorsque vous pilotez l'interpréteur *Tcl* dans un environnement où vous ne voulez pas créer des fenêtres de haut niveau supplémentaires, ou alors si c'est impossible (comme les systèmes Unix/Linux sans un serveur X). Un objet créé par `Tcl()` peut avoir une fenêtre de haut niveau créée (et le sous-système *Tk* initialisé) en appelant sa méthode `loadtk()`.

Parmi les modules qui savent gérer *Tk*, nous pouvons citer :

tkinter
Module *Tkinter* principal

tkinter.colorchooser
Boîte de dialogue permettant à l'utilisateur de choisir une couleur.

tkinter.commondialog
Classe de base pour les boîtes de dialogue définies dans les autres modules listés ici.

tkinter.filedialog
Boîtes de dialogue standard permettant à l'utilisateur de spécifier un fichier à ouvrir ou à enregistrer.

tkinter.font
Utilitaires pour gérer les polices de caractères.

tkinter.messagebox
Accès aux boîtes de dialogue *Tk* standard.

tkinter.scrolledtext
Outil d'affichage de texte avec une barre de défilement verticale intégrée.

tkinter.simpledialog
Boîtes de dialogue simples et fonctions utilitaires.

tkinter.ttk
Ensemble de widgets thématiques introduit dans *Tk* 8.5, offrant des alternatives modernes à de nombreux widgets classiques du module principal `tkinter`.

Modules supplémentaires :

_tkinter
Module binaire qui contient l'interface de bas niveau vers *Tcl/Tk*. Il est automatiquement importé par le module principal `tkinter` et ne doit jamais être utilisé directement par les programmeurs d'applications. Il s'agit généralement d'une bibliothèque partagée (ou *DLL*), mais peut dans certains cas être lié statiquement à l'interpréteur Python.

idlelib
Environnement de développement et d'apprentissage intégré de Python (*IDLE* pour *Integrated Development and Learning Environment*). Basé sur `tkinter`.

tkinter.constants
Constantes symboliques pouvant être utilisées à la place des chaînes lors de la transmission de divers paramètres aux appels *Tkinter*. Importé automatiquement par le module principal `tkinter`.

tkinter.dnd
Gestion du glisser-déposer pour `tkinter`. Il s'agit d'une méthode expérimentale qui ne sera plus maintenue lorsqu'elle sera remplacée par *Tk DND*.

turtle
Tortue graphique dans une fenêtre *Tk*.

26.1.3 Guide de survie *Tkinter*

Cette section n'est pas conçue pour être un tutoriel exhaustif sur *Tk* ou *Tkinter*. Pour cela, reportez-vous à l'une des ressources externes mentionnées précédemment. Cette section fournit plutôt un guide très rapide sur ce à quoi ressemble une application *Tkinter*, identifie les concepts fondamentaux de *Tk* et explique comment l'enveloppe *Tkinter* est structurée.

Le reste de cette section vous aidera à identifier les classes, méthodes et options dont vous aurez besoin dans votre application *Tkinter*, et où trouver une documentation plus détaillée à leur sujet, y compris dans le manuel de référence officiel *Tcl/Tk*.

Un simple programme *Hello World*

Nous commençons par disséquer une application *Hello World* dans *Tkinter*. Ce n'est pas le plus petit que nous puissions écrire, mais il en contient suffisamment pour illustrer certains concepts clés que vous devez connaître.

```
from tkinter import *
from tkinter import ttk
root = Tk()
frm = ttk.Frame(root, padding=10)
frm.grid()
ttk.Label(frm, text="Hello World!").grid(column=0, row=0)
ttk.Button(frm, text="Quit", command=root.destroy).grid(column=1, row=0)
root.mainloop()
```

Après les importations, la ligne suivante crée une instance de la classe `Tk`, qui initialise *Tk* et crée son interpréteur *Tcl* associé. Elle crée également une fenêtre de niveau supérieur, connue sous le nom de fenêtre racine, qui sert de fenêtre principale de l'application.

La ligne suivante crée un widget de cadre, qui dans ce cas contiendra une étiquette et un bouton que nous créerons ensuite. Le cadre est ajusté à l'intérieur de la fenêtre racine.

La ligne suivante crée un widget d'étiquette contenant une chaîne de texte statique. La méthode `grid()` est utilisée pour spécifier la disposition relative (position) de l'étiquette dans son widget cadre contenant, similaire au fonctionnement des tableaux en HTML.

Un widget de bouton est alors créé et placé à droite de l'étiquette. Lorsqu'il est pressé, il appelle la méthode `destroy()` de la fenêtre racine.

Enfin, la méthode `mainloop()` affiche tout sur l'écran et répond aux entrées de l'utilisateur jusqu'à ce que le programme se termine.

Concepts importants de *Tk*

Même ce programme simple illustre les concepts clés suivants de *Tk* :

widgets
: Une interface utilisateur *Tkinter* est composée de *widgets* individuels. Chaque widget est représenté comme un objet Python, instancié à partir de classes comme `ttk.Frame`, `ttk.Label` et `ttk.Button`.

hiérarchie des widgets
: Les widgets sont organisés dans une *hiérarchie*. L'étiquette et le bouton étaient contenus dans un cadre, qui à son tour était contenu dans la fenêtre racine. Lors de la création de chaque widget *enfant*, son widget *parent* est passé comme premier argument au constructeur du widget.

options de configuration
: Les widgets ont des *options de configuration*, qui modifient leur apparence et leur comportement, comme le texte à afficher dans une étiquette ou un bouton. Différentes classes de widgets auront différents ensembles d'options.

gestion de la géométrie
: Les widgets ne sont pas automatiquement ajoutés à l'interface utilisateur lorsqu'ils sont créés. Un *gestionnaire de géométrie* comme `grid` contrôle où ils sont placés dans l'interface utilisateur.

boucle d'événements
: *Tkinter* réagit aux entrées de l'utilisateur, aux modifications de votre programme et même rafraîchit l'affichage uniquement lors de l'exécution active d'une *boucle d'événements*. Si votre programme n'exécute pas la boucle d'événements, votre interface utilisateur n'est pas mise à jour.

Comprendre comment *Tkinter* enveloppe *Tcl/Tk*

When your application uses Tkinter's classes and methods, internally Tkinter is assembling strings representing Tcl/Tk commands, and executing those commands in the Tcl interpreter attached to your application's `Tk` instance.

Qu'il s'agisse d'essayer de naviguer dans la documentation de référence, d'essayer de trouver la bonne méthode ou option, d'adapter du code existant ou de déboguer votre application *Tkinter*, il arrive qu'il soit utile de comprendre à quoi ressemblent ces commandes *Tcl/Tk* sous-jacentes.

Pour illustrer cette idée, voici l'équivalent *Tcl/Tk* de la partie principale du script *Tkinter* ci-dessus.

```
ttk::frame .frm -padding 10
grid .frm
grid [ttk::label .frm.lbl -text "Hello World!"] -column 0 -row 0
grid [ttk::button .frm.btn -text "Quit" -command "destroy ."] -column 1 -row 0
```

La syntaxe de *Tcl* est similaire à celle de nombreux langages *shell*, où le premier mot est la commande à exécuter, suivi des arguments de cette commande, séparés par des espaces. Sans entrer dans trop de détails, notez ce qui suit :

- Les commandes utilisées pour créer des widgets (comme `ttk::frame`) correspondent aux classes de widgets dans *Tkinter*.
- Les options du widget *Tcl* (comme `-text`) correspondent aux arguments de mots-clés dans *Tkinter*.
- Les widgets sont référencés par un *chemin* en *Tcl* (comme `.frm.btn`), alors que *Tkinter* n'utilise pas de noms mais des références d'objets.
- La place d'un widget dans la hiérarchie des widgets est encodée dans son nom de chemin (hiérarchique), qui utilise un `.` (point) comme séparateur de chemin. Le chemin d'accès à la fenêtre racine est simplement `.` (point). Dans *Tkinter*, la hiérarchie n'est pas définie par le nom de chemin mais en spécifiant le widget parent lors de la création de chaque widget enfant.
- Les opérations qui sont implémentées comme des *commandes* séparées dans *Tcl* (comme `grid` ou `destroy`) sont représentées comme des *méthodes* sur les objets widget de *Tkinter*. Comme vous le verrez bientôt, à d'autres moments, *Tcl* utilise ce qui semble être des appels de méthode sur des objets widget, qui reflètent plus fidèlement ce qui serait utilisé dans *Tkinter*.

Comment puis-je... ? Quelle option... ?

Si vous ne savez pas comment faire quelque chose dans *Tkinter* et que vous ne le trouvez pas immédiatement dans le didacticiel ou la documentation de référence que vous utilisez, il existe quelques stratégies qui peuvent être utiles.

Tout d'abord, rappelez-vous que les détails du fonctionnement des widgets individuels peuvent varier selon les différentes versions de *Tkinter* et de *Tcl/Tk*. Si vous recherchez de la documentation, assurez-vous qu'elle correspond aux versions Python et *Tcl/Tk* installées sur votre système.

Lorsque vous recherchez comment utiliser une API, il est utile de connaître le nom exact de la classe, de l'option ou de la méthode que vous utilisez. L'introspection, soit dans un shell Python interactif, soit avec `print()`, peut vous aider à identifier ce dont vous avez besoin.

Pour savoir quelles options de configuration sont disponibles sur n'importe quel widget, appelez sa méthode `configure()`, qui renvoie un dictionnaire contenant une variété d'informations sur l'objet, y compris ses valeurs par défaut et actuelles. Utilisez `keys()` pour obtenir uniquement les noms de chaque option.

```
btn = ttk.Button(frm, ...)
print(btn.configure().keys())
```

Comme la plupart des widgets ont de nombreuses options de configuration en commun, il peut être utile de savoir lesquelles sont spécifiques à une classe de widget particulière. Comparer la liste des options à celle d'un widget plus simple, comme un cadre, est une façon de faire.

```
print(set(btn.configure().keys()) - set(frm.configure().keys()))
```

De même, vous pouvez trouver les méthodes disponibles pour un objet widget en utilisant la fonction standard `dir()`. Si vous l'essayez, vous verrez qu'il existe plus de 200 méthodes de widget courantes donc, encore une fois, identifier celles spécifiques à une classe de widget est utile.

```
print(dir(btn))
print(set(dir(btn)) - set(dir(frm)))
```

Navigation dans le manuel de référence *Tcl/Tk*

Comme indiqué, le manuel de référence officiel des commandes Tk (pages de manuel) est souvent la description la plus précise sur ce que font des opérations spécifiques sur les widgets. Même lorsque vous connaissez le nom de l'option ou de la méthode dont vous avez besoin, il se peut que vous ayez encore quelques endroits à chercher.

Alors que toutes les opérations dans *Tkinter* sont implémentées en tant qu'appels de méthode sur des objets widget, vous avez vu que de nombreuses opérations *Tcl/Tk* apparaissent sous forme de commandes qui prennent un chemin d'accès de widget comme premier paramètre, suivi de paramètres facultatifs, par exemple :

```
destroy .
grid .frm.btn -column 0 -row 0
```

D'autres, cependant, ressemblent plus à des méthodes appelées sur un objet widget (en fait, lorsque vous créez un widget en *Tcl/Tk*, il crée une commande *Tcl* avec le nom du chemin du widget, le premier paramètre de cette commande étant le nom d'une méthode à appeler).

```
.frm.btn invoke
.frm.lbl configure -text "Goodbye"
```

Dans la documentation de référence officielle *Tcl/Tk*, vous trouvez la plupart des opérations qui ressemblent à des appels de méthode dans la page de manuel d'un widget spécifique (par exemple, vous trouvez la méthode `invoke()` dans la page de manuel sur le ttk ::button), tandis que les fonctions qui prennent un widget comme paramètre ont souvent leur propre page de manuel (par exemple, grid).

Vous trouvez de nombreuses options et méthodes courantes dans les pages de manuel des options ou ttk ::widget, tandis que d'autres se trouvent dans la page de manuel d'une classe de widget spécifique.

Vous constaterez également que de nombreuses méthodes *Tkinter* ont des noms composés, par exemple `winfo_x()`, `winfo_height()`, `winfo_viewable()`. Vous trouverez de la documentation pour tout cela dans la page de manuel winfo.

> **Note**
>
> de manière quelque peu déroutante, il existe également des méthodes sur tous les widgets *Tkinter* qui ne fonctionnent pas réellement sur le widget, mais fonctionnent à une portée globale, indépendamment de tout widget. Les méthodes d'accès au presse-papiers ou à la sonnerie du système en font partie (il se trouve qu'elles sont implémentées en tant que méthodes dans la classe de base `Widget` dont héritent tous les widgets *Tkinter*).

26.1.4 Fils d'exécution multiples

Python et *Tcl/Tk* gèrent les fils d'exécution multiples (*threads*) de manière très différente, et `tkinter` essaie de combler cette différence. Si vous utilisez des fils d'exécution multiples, vous devrez sûrement en être conscient.

Un interpréteur Python peut être associé à de nombreux fils d'exécution. En *Tcl*, plusieurs fils d'exécution peuvent être créés, mais chaque fil d'exécution est associé à une instance d'interpréteur *Tcl* distincte. Les fils d'exécution multiples peuvent également créer plusieurs instances d'interpréteur, bien que chaque instance d'interpréteur ne puisse être utilisée que par le seul fil d'exécution qui l'a créée.

Chaque objet `Tk` créé par `tkinter` contient un interpréteur *Tcl*. Il garde également une trace du fil d'exécution qui a créé cet interpréteur. Les appels à `tkinter` peuvent être effectués à partir de n'importe quel fil d'exécution Python. En interne, si un appel provient d'un fil d'exécution autre que celui qui a créé l'objet `Tk`, un événement est posté dans la file d'attente d'événements de l'interpréteur et, lorsqu'il est exécuté, le résultat est renvoyé au fil d'exécution Python appelant.

Les applications *Tcl/Tk* sont normalement pilotées par les événements, ce qui signifie qu'après l'initialisation, l'interpréteur exécute une boucle d'événements (c'est-à-dire `Tk.mainloop()`) et répond aux événements. Comme il s'agit d'un seul fil d'exécution, les gestionnaires d'événements doivent répondre rapidement, sinon ils empêchent le traitement d'autres événements. Pour éviter cela, les calculs de longue durée ne doivent pas s'exécuter dans un gestionnaire d'événements, mais sont soit divisés en plus petits morceaux à l'aide de temporisateurs, soit exécutés dans un autre

fil d'exécution. Ceci est différent de nombreux kits d'outils d'interface graphique où l'interface graphique s'exécute dans un fil d'exécution complètement séparé de tout le code d'application, y compris les gestionnaires d'événements.

Si l'interpréteur *Tcl* n'exécute pas la boucle d'événements et ne traite pas les événements, tout appel `tkinter` effectué à partir de fils d'exécution autres que celui exécutant l'interpréteur *Tcl* échoue.

Plusieurs cas particuliers existent :

- Les bibliothèques *Tcl/Tk* peuvent être construites de manière à ne pas gérer les fils d'exécution multiples. Dans ce cas, `tkinter` appelle la bibliothèque à partir du fil d'exécution Python d'origine, même s'il est différent du fil d'exécution qui a créé l'interpréteur *Tcl*. Un verrou global garantit qu'un seul appel se produit à la fois.
- Alors que `tkinter` vous permet de créer plus d'une instance d'un objet `Tk` (avec son propre interpréteur), tous les interpréteurs qui font partie du même fil d'exécution partagent une file d'attente d'événements commune, qui devient très vite moche. En pratique, ne créez pas plus d'une instance de `Tk` à la fois. Sinon, il est préférable de les créer dans des fils d'exécution séparés et de vous assurer que vous exécutez une instance *Tcl/Tk* compatible avec les fils d'exécution multiples.
- Le blocage des gestionnaires d'événements n'est pas le seul moyen d'empêcher l'interpréteur *Tcl* de bien gérer la boucle d'événements. Il est même possible d'exécuter plusieurs boucles d'événements imbriquées ou d'abandonner complètement la boucle d'événements. Si vous faites quelque chose de délicat en ce qui concerne les événements ou les fils d'exécution, soyez conscient de ces possibilités.
- Il existe quelques fonctions select `tkinter` qui ne fonctionnent actuellement que lorsqu'elles sont appelées depuis le fil d'exécution qui a créé l'interpréteur *Tcl*.

26.1.5 Guide pratique

Définition des options

Les options contrôlent des paramètres tels que la couleur et la largeur de la bordure d'un objet graphique. Les options peuvent être réglées de trois façons :

Lors de la création de l'objet, à l'aide d'arguments par mots-clés

```
fred = Button(self, fg="red", bg="blue")
```

Après la création de l'objet, en manipulant le nom de l'option comme une entrée de dictionnaire

```
fred["fg"] = "red"
fred["bg"] = "blue"
```

Utilisez la méthode `config()` pour mettre à jour plusieurs attributs après la création de l'objet

```
fred.config(fg="red", bg="blue")
```

Pour l'explication complète d'une option donnée et de son comportement, voir les pages de manuel *Tk* de l'objet graphique en question.

Notez que les pages de manuel listent « OPTIONS STANDARD » et « OPTIONS SPÉCIFIQUES D'OBJETS GRAPHIQUES » pour chaque objet graphique. La première est une liste d'options communes à de nombreux objets graphiques, la seconde est une liste d'options propres à cet objet graphique particulier. Les options standard sont documentées sur la page de manuel `options(3)`.

Aucune distinction n'est faite dans ce document entre les options standard et les options spécifiques à un objet graphique. Certaines options ne s'appliquent pas à certains types d'objets graphiques. La réaction d'un objet graphique donné à une option particulière dépend de la classe de l'objet graphique ; les boutons possèdent une option `command`, pas les étiquettes.

Les options gérées par un objet graphique donné sont listées dans la page de manuel de cet objet graphique, ou peuvent être interrogées à l'exécution en appelant la méthode `config()` sans argument, ou en appelant la méthode `keys()` sur cet objet graphique. La valeur de retour de ces appels est un dictionnaire dont la clé est le nom de l'option sous forme de chaîne (par exemple, `'relief'`) et dont les valeurs sont des *5-uplets*.

Certaines options, comme `bg`, sont des synonymes d'options communes qui ont des noms longs (`bg` est une abréviation pour `background` « arrière-plan »). Passer le nom abrégé d'une option à la méthode `config()` renvoie un couple, pas un quintuplet. Le couple renvoyé contient le nom abrégé et le nom *réel* de l'option, par exemple `('bg','background')`.

Index	Signification	Exemple
0	nom des options	`'relief'`
1	nom de l'option pour la recherche dans la base de données	`'relief'`
2	classe de l'option pour la recherche dans la base de données	`'Relief'`
3	valeur par défaut	`'raised'`
4	valeur actuelle	`'groove'`

Exemple :

```
>>> print(fred.config())
{'relief': ('relief', 'relief', 'Relief', 'raised', 'groove')}
```

Bien sûr, le dictionnaire affiché contient toutes les options disponibles et leurs valeurs. Ceci n'est donné qu'à titre d'exemple.

L'empaqueteur

L'empaqueteur est l'un des mécanismes de *Tk* pour la gestion de la disposition des éléments sur l'écran. Les gestionnaires de géométrie sont utilisés pour spécifier le positionnement relatif du positionnement des objets graphiques dans leur conteneur — leur *constructeur* mutuel. Contrairement au plus encombrant *placeur* (qui est utilisé moins souvent, et nous n'en parlons pas ici), l'empaqueteur prend les spécifications qualitatives de relation — *above*, *to the left of*, *filling*, etc — et calcule tout pour déterminer les coordonnées exactes du placement pour vous.

La taille d'un objet graphique *constructeur* est déterminée par la taille des « objets graphiques hérités » à l'intérieur. L'empaqueteur est utilisé pour contrôler l'endroit où les objets graphiques hérités apparaissent à l'intérieur du constructeur dans lequel ils sont empaquetés. Vous pouvez regrouper des objets graphiques dans des cadres, et des cadres dans d'autres cadres, afin d'obtenir le type de mise en page souhaité. De plus, l'arrangement est ajusté dynamiquement pour s'adapter aux changements incrémentiels de la configuration, une fois qu'elle est empaquetées.

Notez que les objets graphiques n'apparaissent pas tant que leur disposition n'a pas été spécifiée avec un gestionnaire de géométrie. C'est une erreur de débutant courante de ne pas tenir compte de la spécification de la géométrie, puis d'être surpris lorsque l'objet graphique est créé mais que rien n'apparaît. Un objet graphique n'apparaît qu'après que, par exemple, la méthode `pack()` de l'empaqueteur lui ait été appliquée.

La méthode `pack()` peut être appelée avec des paires mot-clé-option/valeur qui contrôlent où l'objet graphique doit apparaître dans son conteneur et comment il doit se comporter lorsque la fenêtre principale de l'application est redimensionnée. En voici quelques exemples :

```
fred.pack()                     # defaults to side = "top"
fred.pack(side="left")
fred.pack(expand=1)
```

Options de l'empaqueteur

Pour de plus amples informations sur l'empaqueteur et les options qu'il peut prendre, voir les pages de manuel et la page 183 du livre de John Ousterhout.

anchor
: Type d'ancrage. Indique l'endroit où l'empaqueteur doit placer chaque enfant dans son espace.

expand
: Booléen, `0` ou `1`.

fill
: Valeurs acceptées : `'x'`, `'y'`, `'both'`, `'none'`.

ipadx* et *ipady
: Une distance — désignant l'écart interne de chaque côté de l'objet graphique hérité.

padx* et *pady
: Une distance — désignant l'écart externe de chaque côté de l'objet graphique hérité.

side
: Valeurs acceptées : `'left'`, `'right'`, `'top'`, `'bottom'`.

Association des variables de l'objet graphique

L'assignation d'une valeur à certains objets graphiques (comme les objets graphique de saisie de texte) peut être liée directement aux variables de votre application à l'aide d'options spéciales. Ces options sont `variable`, `textvariable`, `onvalue`, `offvalue` et `value`. Ce lien fonctionne dans les deux sens : si la variable change pour une raison ou pour une autre, l'objet graphique auquel elle est connectée est mis à jour pour refléter la nouvelle valeur.

Malheureusement, dans l'implémentation actuelle de *tkinter* il n'est pas possible de passer une variable Python arbitraire à un objet graphique via une option `variable` ou `textvariable`. Les seuls types de variables pour lesquels cela fonctionne sont les variables qui sont sous-classées à partir d'une classe appelée `Variable`, définie dans *tkinter*.

Il existe de nombreuses sous-classes utiles de `Variable` déjà définies : `StringVar`, `IntVar`, `DoubleVar` et `BooleanVar`. Pour lire la valeur courante d'une telle variable, appelez la méthode `get()` dessus et, pour changer sa valeur, appelez la méthode `set()`. Si vous suivez ce protocole, l'objet graphique suivra toujours la valeur de la variable, sans autre intervention de votre part.

Par exemple :

```
import tkinter as tk

class App(tk.Frame):
    def __init__(self, master):
        super().__init__(master)
        self.pack()

        self.entrythingy = tk.Entry()
        self.entrythingy.pack()

        # Create the application variable.
        self.contents = tk.StringVar()
        # Set it to some value.
        self.contents.set("this is a variable")
        # Tell the entry widget to watch this variable.
        self.entrythingy["textvariable"] = self.contents

        # Define a callback for when the user hits return.
        # It prints the current value of the variable.
        self.entrythingy.bind('<Key-Return>',
                              self.print_contents)

    def print_contents(self, event):
        print("Hi. The current entry content is:",
              self.contents.get())

root = tk.Tk()
myapp = App(root)
myapp.mainloop()
```

Le gestionnaire de fenêtres

Dans *Tk*, il y a une commande pratique, `wm`, pour interagir avec le gestionnaire de fenêtres. Les options de la commande `wm` vous permettent de contrôler les titres, le placement, les icônes en mode *bitmap* et encore d'autres choses du même genre. Dans *tkinter*, ces commandes ont été implémentées en tant que méthodes sur la classe `Wm`. Les objets graphiques de haut niveau sont sous-classés à partir de la classe `Wm`, ils peuvent donc appeler directement les méthodes de `Wm`.

Pour accéder à la fenêtre du plus haut niveau qui contient un objet graphique donné, vous pouvez souvent simplement vous référer au parent de cet objet graphique. Bien sûr, si l'objet graphique a été empaqueté à l'intérieur d'un cadre, le parent ne représentera pas la fenêtre de plus haut niveau. Pour accéder à la fenêtre du plus haut niveau qui contient un

objet graphique arbitraire, vous pouvez appeler la méthode `_root()`. Cette méthode commence par un soulignement pour indiquer que cette fonction fait partie de l'implémentation, et non d'une interface avec la fonctionnalité *Tk*.

Voici quelques exemples d'utilisation courante :

```
import tkinter as tk

class App(tk.Frame):
    def __init__(self, master=None):
        super().__init__(master)
        self.pack()

# create the application
myapp = App()

#
# here are method calls to the window manager class
#
myapp.master.title("My Do-Nothing Application")
myapp.master.maxsize(1000, 400)

# start the program
myapp.mainloop()
```

Types de données des options *Tk*

anchor

Les valeurs acceptées sont des points cardinaux : `"n"`, `"ne"`, `"e"`, `"se"`, `"s"`, `"sw"`, `"w"`, `"nw"` et `"center"`.

bitmap

Il y a huit bitmaps intégrés nommés : `"error"`, `"gray25"`, `"gray50"`, `"hourglass"`, `"info"`, `"questhead"`, `"question"`, `"warning"`. Pour spécifier un nom de fichier bitmap X, indiquez le chemin complet du fichier, précédé de `@`, comme dans `"@/usr/contrib/bitmap/gumby.bit"`.

boolean

Vous pouvez lui donner les entiers 0 ou 1 ou les chaînes de caractères `"yes"` ou `"no"`.

callback

N'importe quelle fonction Python qui ne prend pas d'argument. Par exemple :

```
def print_it():
    print("hi there")
fred["command"] = print_it
```

color

Les couleurs peuvent être données sous forme de noms de couleurs *Xorg* dans le fichier *rgb.txt*, ou sous forme de chaînes représentant les valeurs RVB en 4 bits : `"#RGB"`, 8 bits : `"#RRVVBB"`, 12 bits : `"#RRRVVVBBB"`, ou 16 bits : `"#RRRRVVVVBBBB"`, où R,V,B représente ici tout chiffre hexadécimal valide. Voir page 160 du livre d'Ousterhout pour plus de détails.

cursor

Les noms de curseurs *Xorg* standard que l'on trouve dans `cursorfont.h` peuvent être utilisés, sans le préfixe `XC_`. Par exemple pour obtenir un curseur en forme de main (`XC_hand2`), utilisez la chaîne `"hand2"`. Vous pouvez également spécifier votre propre bitmap et fichier masque. Voir page 179 du livre d'Ousterhout.

distance

Les distances à l'écran peuvent être spécifiées en pixels ou en distances absolues. Les pixels sont donnés sous forme de nombres et les distances absolues sous forme de chaînes de caractères, le dernier caractère indiquant les unités : `c` pour les centimètres, `i` pour les pouces (*inches* en anglais), `m` pour les millimètres, `p` pour les points d'impression. Par exemple, 3,5 pouces est noté `"3.5i"`.

font

Tk utilise un format de nom de police sous forme de liste, tel que `{courier 10 bold}`. Les tailles de

polices avec des nombres positifs sont mesurées en points ; les tailles avec des nombres négatifs sont mesurées en pixels.

geometry
Il s'agit d'une chaîne de caractères de la forme `largeurxhauteur`, où la largeur et la hauteur sont mesurées en pixels pour la plupart des objets graphiques (en caractères pour les objets graphiques affichant du texte). Par exemple : `fred["geometry"] = "200x100"`.

justify
Les valeurs acceptées sont les chaînes de caractères : `"left"`, `"center"`, `"right"` et `"fill"`.

region
c'est une chaîne de caractères avec quatre éléments séparés par des espaces, chacun d'eux étant une distance valide (voir ci-dessus). Par exemple : `"2 3 4 5"`, `" 3i 2i 4.5i 2i"` et `"3c 2c 4c 10.43c"` sont toutes des régions valides.

relief
Détermine le style de bordure d'un objet graphique. Les valeurs valides sont : `"raised"`, `"sunken"`, `"flat"`, `"groove"` et `"ridge"`.

scrollcommand
C'est presque toujours la méthode `set()` d'un objet graphique de défilement, mais peut être n'importe quelle méthode d'objet graphique qui prend un seul argument.

wrap
Doit être `"none"`, `"char"` ou `"word"`.

Liaisons et événements

La méthode `bind` de la commande d'objet graphique vous permet de surveiller certains événements et d'avoir un déclencheur de fonction de rappel lorsque ce type d'événement se produit. La forme de la méthode de liaison est la suivante :

```
def bind(self, sequence, func, add=''):
```

où :

sequence
est une chaîne de caractères qui indique le type d'événement cible. (Voir la page du manuel de `bind(3tk)` et la page 201 du livre de John Ousterhout *Tcl and the Tk Toolkit (2nd edition)* pour plus de détails).

func
est une fonction Python, prenant un argument, à invoquer lorsque l'événement se produit. Une instance d'évènement sera passée en argument. (Les fonctions déployées de cette façon sont communément appelées *callbacks* ou « fonctions de rappel » en français).

add
est facultative, soit `''` ou `'+'`. L'envoi d'une chaîne de caractères vide indique que cette liaison doit remplacer toute autre liaison à laquelle cet événement est associé. L'envoi de `"+"` signifie que cette fonction doit être ajoutée à la liste des fonctions liées à ce type d'événement.

Par exemple :

```
def turn_red(self, event):
    event.widget["activeforeground"] = "red"

self.button.bind("<Enter>", self.turn_red)
```

Remarquez comment on accède au champ *objet graphique* de l'événement dans la fonction de rappel `turn_red()`. Ce champ contient l'objet graphique qui a capturé l'événement *Xorg*. Le tableau suivant répertorie les autres champs d'événements auxquels vous pouvez accéder, et comment ils sont nommés dans *Tk*, ce qui peut être utile lorsque vous vous référez aux pages de manuel *Tk*.

Tk	Champ *évènement* de *Tkinter*	Tk	Champ *évènement* de *Tkinter*
%f	focus	%A	char
%h	hauteur	%E	send_event
%k	keycode	%K	keysym
%s	state	%N	keysym_num
%t	time	%T	type
%w	width	%W	widget
%x	x	%X	x_root
%y	y	%Y	y_root

Le paramètre index

Un certain nombre d'objets graphiques nécessitent le passage de paramètres « indicés ». Ils sont utilisés pour pointer vers un endroit spécifique dans un objet graphique de type *Texte*, ou vers des caractères particuliers dans un objet graphique de type *Entrée*, ou vers des éléments de menu particuliers dans un objet graphique de type *Menu*.

Index des objets graphique de type *Entrée* (`index`, `view index`, etc.)
Les objets graphiques de type *Entrée* ont des options qui se réfèrent à la position des caractères dans le texte affiché. Vous pouvez utiliser ces fonctions `tkinter` pour accéder à ces points spéciaux dans les widgets textes :

Index des objets graphiques texte
La notation de l'index des objets graphiques de type *Texte* est très riche et mieux décrite dans les pages du manuel *Tk*.

Index menu (`menu.invoke()`, `menu.entryconfig()`, etc.)
Certaines options et méthodes pour manipuler les menus nécessitent des éléments de spécifiques. Chaque fois qu'un index de menu est nécessaire pour une option ou un paramètre, vous pouvez utiliser :
- un entier qui fait référence à la position numérique de l'entrée dans l'objet graphique, comptée à partir du haut, en commençant par 0 ;
- la chaîne de caractères `"active"`, qui fait référence à la position du menu qui se trouve actuellement sous le curseur ;
- la chaîne de caractères `"last"` qui fait référence au dernier élément du menu ;
- un entier précédé de `@`, comme dans `@6`, où l'entier est interprété comme une coordonnée y de pixels dans le système de coordonnées du menu ;
- la chaîne de caractères `"none"`, qui n'indique aucune entrée du menu, le plus souvent utilisée avec `menu.activate()` pour désactiver toutes les entrées, et enfin,
- une chaîne de texte dont le motif correspond à l'étiquette de l'entrée de menu, telle qu'elle est balayée du haut vers le bas du menu. Notez que ce type d'index est considéré après tous les autres, ce qui signifie que les correspondances pour les éléments de menu étiquetés `last`, `active` ou `none` peuvent être interprétés comme les littéraux ci-dessus, plutôt.

Images

Des images de différents formats peuvent être créées à travers la sous-classe correspondante de `tkinter.Image` :
- `BitmapImage` pour les images au format *XBM*.
- `PhotoImage` pour les images aux formats *PGM*, *PPM*, *GIF* et *PNG*. Ce dernier est géré à partir de *Tk* 8.6.

L'un ou l'autre type d'image est créé par l'option `file` ou `data` (d'autres options sont également disponibles).

Modifié dans la version 3.13 : Added the `PhotoImage` method `copy_replace()` to copy a region from one image to other image, possibly with pixel zooming and/or subsampling. Add *from_coords* parameter to `PhotoImage` methods `copy()`, `zoom()` and `subsample()`. Add *zoom* and *subsample* parameters to `PhotoImage` method `copy()`.

L'objet image peut alors être utilisé partout où un objet graphique sait gérer une option `image` (par ex. étiquettes, boutons, menus). Dans ces cas, *Tk* ne conserve pas de référence à l'image. Lorsque la dernière référence Python à l'objet image est supprimée, les données de l'image sont également supprimées, et *Tk* affiche une boite vide à l'endroit où l'image était utilisée.

Voir aussi

Le paquet Pillow ajoute la prise en charge de formats tels que *BMP*, *JPEG*, *TIFF* et *WebP*, entre autres.

26.1.6 Gestionnaires de fichiers

Tk vous permet d'enregistrer et de *désenregistrer* une fonction de rappel qui est appelée depuis la boucle principale de *Tk* lorsque des entrées-sorties sont possibles sur un descripteur de fichier. Un seul gestionnaire peut être enregistré par descripteur de fichier. Exemple de code :

```
import tkinter
widget = tkinter.Tk()
mask = tkinter.READABLE | tkinter.WRITABLE
widget.tk.createfilehandler(file, mask, callback)
...
widget.tk.deletefilehandler(file)
```

Cette fonction n'est pas disponible sous Windows.

Dans la mesure où vous ne savez pas combien d'octets sont disponibles en lecture, il ne faut pas utiliser les méthodes `BufferedIOBase` ou `TextIOBase read()` ou `readline()`, car elles requièrent d'indiquer le nombre de *bytes* à lire. Pour les connecteurs, les méthodes `recv()` ou `recvfrom()` fonctionnent bien ; pour les autres fichiers, utilisez des lectures brutes ou `os.read(file.fileno(), maxbytecount)`.

`Widget.tk.`**`createfilehandler`**(*file*, *mask*, *func*)

Enregistre la fonction de rappel du gestionnaire de fichiers *func*. L'argument `file` peut être soit un objet avec une méthode `fileno()` (comme un objet fichier ou connecteur), soit un descripteur de fichier de type entier. L'argument `mask` est une combinaison *OU* de l'une des trois constantes ci-dessous. La fonction de rappel s'utilise comme suit :

```
callback(file, mask)
```

`Widget.tk.`**`deletefilehandler`**(*file*)

Désenregistre un gestionnaire de fichiers.

`_tkinter.`**`READABLE`**

`_tkinter.`**`WRITABLE`**

`_tkinter.`**`EXCEPTION`**

Constantes utilisées dans les arguments `mask`.

26.2 `tkinter.colorchooser` --- Color choosing dialog

Code source : Lib/tkinter/colorchooser.py

Le module `tkinter.colorchooser` fournit la classe `Chooser` en tant qu'interface avec la boîte de dialogue native du sélecteur de couleurs. `Chooser` implémente une fenêtre de dialogue de choix de couleur modale. La classe `Chooser` hérite de la classe `Dialog`.

class `tkinter.colorchooser.`**`Chooser`**(*master=None*, ***options*)

`tkinter.colorchooser.`**`askcolor`**(*color=None*, ***options*)

Crée une boîte de dialogue de choix de couleur. Un appel à cette méthode affiche la fenêtre, attend que l'utilisateur fasse une sélection et renvoie la couleur sélectionnée (ou `None`) à l'appelant.

Voir aussi

Module `tkinter.commondialog`
Module de dialogue standard *Tkinter*

26.3 `tkinter.font` --- Tkinter font wrapper

Code source : Lib/tkinter/font.py

Le module `tkinter.font` fournit la classe `Font` pour créer et utiliser des polices nommées.

Les différentes épaisseurs et inclinaisons des polices sont :

`tkinter.font.`**`NORMAL`**

`tkinter.font.`**`BOLD`**

`tkinter.font.`**`ITALIC`**

`tkinter.font.`**`ROMAN`**

class `tkinter.font.`**`Font`**(*root=None*, *font=None*, *name=None*, *exists=False*, ***options*)

La classe `Font` représente une police nommée. Les instances *Font* reçoivent des noms uniques et peuvent être spécifiées par leur configuration de famille, de taille et de style. Les polices nommées sont la méthode de *Tk* pour créer et identifier les polices comme un seul objet, plutôt que de spécifier une police par ses attributs à chaque occurrence.

arguments :

font – *n*-uplet spécificateur de police (famille, taille, options)
name – nom de police unique
exists – s'il est vrai, *self* pointe vers la police nommée existante

options nommées supplémentaires (ignorées si *font* est spécifié) :

family – famille de polices, c'est-à-dire *Courier*, *Times*
size – taille de la police
Si *size* est positif, il est interprété comme une taille en points.
Si *size* est un nombre négatif sa valeur absolue est traitée comme taille en pixels.
weight – accentuation de la police (*NORMAL*, *BOLD* pour gras)
slant – *ROMAN* pour romain, *ITALIC* pour italique
underline – soulignement de la police (0 – aucun, 1 – souligné)
overstrike – police barrée (0 – aucune, 1 – barré)

actual(*option=None*, *displayof=None*)

Renvoie les attributs de la police.

cget(*option*)

Récupère un attribut de la police.

config(***options*)

Modifie les attributs de la police.

copy()

Renvoie une nouvelle instance de la police actuelle.

measure(*text*, *displayof=None*)

Renvoie la quantité d'espace que le texte occuperait sur l'affichage spécifié s'il était formaté dans la police actuelle. Si *displayof* n'est pas spécifié, *tk* suppose que c'est la fenêtre principale de l'application.

metrics(**options*, ***kw*)

Renvoie des données spécifiques à la police. Les options incluent :

ascent – **distance entre la ligne de base et le point le plus haut qu'un** caractère de la police peut occuper

descent – **distance entre la ligne de base et le point le plus bas qu'un** caractère de la police peut occuper

linespace – **séparation verticale minimale nécessaire entre deux** caractères de la police qui assure l'absence de chevauchement vertical entre les lignes.

fixed – 1 si la police est à largeur fixe sinon 0

tkinter.font.**families**(*root=None*, *displayof=None*)

Renvoie les différentes familles de polices.

tkinter.font.**names**(*root=None*)

Renvoie les noms des polices définies.

tkinter.font.**nametofont**(*name*, *root=None*)

Renvoie une représentation `Font` d'une police *tk* nommée.

Modifié dans la version 3.10 : le paramètre *root* a été ajouté.

26.4 Boîtes de dialogue *Tkinter*

26.4.1 `tkinter.simpledialog` – Boîtes de dialogue de saisie standard de *Tkinter*

Code source : Lib/tkinter/simpledialog.py

Le module `tkinter.simpledialog` contient des classes pratiques et des fonctions pour créer des boîtes de dialogue modales simples afin d'obtenir une valeur de l'utilisateur.

tkinter.simpledialog.**askfloat**(*title*, *prompt*, ***kw*)

tkinter.simpledialog.**askinteger**(*title*, *prompt*, ***kw*)

tkinter.simpledialog.**askstring**(*title*, *prompt*, ***kw*)

Les trois fonctions ci-dessus fournissent des boîtes de dialogue qui invitent l'utilisateur à saisir une valeur du type souhaité.

class tkinter.simpledialog.**Dialog**(*parent*, *title=None*)

Classe mère pour les boîtes de dialogue personnalisées.

body(*master*)

À remplacer pour construire l'interface de la boîte de dialogue et renvoyer le widget qui doit avoir le focus initial.

buttonbox()

Le comportement par défaut ajoute les boutons OK et Annuler. À remplacer avoir une disposition de boutons personnalisée.

26.4.2 `tkinter.filedialog` – Boîtes de dialogue de sélection de fichiers

Code source : Lib/tkinter/filedialog.py

Le module `tkinter.filedialog` fournit des classes et des fonctions de fabrique pour créer des fenêtres de sélection de fichiers ou répertoires.

Boîtes de dialogue de chargement et sauvegarde natives

Les classes et fonctions suivantes fournissent des fenêtres de dialogue de fichiers qui combinent une apparence native avec des options de configuration pour personnaliser le comportement. Les arguments nommés suivants s'appliquent aux classes et fonctions répertoriées ci-dessous :

parent – la fenêtre sur laquelle placer la boîte de dialogue

title – le titre de la fenêtre

initialdir – le répertoire dans lequel la boîte de dialogue démarre

initialfile – le fichier sélectionné à l'ouverture de la boîte de dialogue

filetypes – une séquence de *n*-uplets (étiquette, motif), le caractère générique `'*'` est autorisé

defaultextension – extension par défaut à ajouter au fichier (boîtes de dialogue de sauvegarde)

multiple – lorsque vrai, la sélection de plusieurs éléments est autorisée

Fonctions statiques de fabrique

Les fonctions ci-dessous, lorsqu'elles sont appelées, créent une boîte de dialogue d'apparence modale native, attendent la sélection de l'utilisateur, puis renvoient la ou les valeurs sélectionnées ou `None` à l'appelant.

`tkinter.filedialog.`**`askopenfile`**(*mode='r'*, ***options*)

`tkinter.filedialog.`**`askopenfiles`**(*mode='r'*, ***options*)

Les deux fonctions ci-dessus créent une boîte de dialogue `Open` et renvoient le(s) objet(s) fichier ouvert(s) en mode lecture seule.

`tkinter.filedialog.`**`asksaveasfile`**(*mode='w'*, ***options*)

Crée une boîte de dialogue `SaveAs` et renvoie un objet fichier ouvert en mode écriture seule.

`tkinter.filedialog.`**`askopenfilename`**(***options*)

`tkinter.filedialog.`**`askopenfilenames`**(***options*)

Les deux fonctions ci-dessus créent une boîte de dialogue `Open` et renvoient le(s) nom(s) de fichier sélectionné(s) qui correspondent au(x) fichier(s) existant(s).

`tkinter.filedialog.`**`asksaveasfilename`**(***options*)

Crée une boîte de dialogue `SaveAs` et renvoie le nom de fichier sélectionné.

`tkinter.filedialog.`**`askdirectory`**(***options*)

Demande à l'utilisateur de sélectionner un répertoire.

Option nommée supplémentaire :

mustexist – détermine si la sélection doit être un répertoire existant.

`class` `tkinter.filedialog.`**`Open`**(*master=None*, ***options*)

`class` `tkinter.filedialog.`**`SaveAs`**(*master=None*, ***options*)

Les deux classes ci-dessus fournissent des fenêtres de dialogue natives pour enregistrer et charger des fichiers.

Classes de commodité

Les classes ci-dessous sont utilisées pour créer des fenêtres avec fichiers et répertoires à partir de zéro. Celles-ci n'imitent pas l'apparence native de la plateforme.

class tkinter.filedialog.**Directory**(*master=None*, ***options*)

Crée une boîte de dialogue invitant l'utilisateur à sélectionner un répertoire.

> **Note**
>
> la classe *FileDialog* doit être sous-classée pour un comportement et une gestion des événements personnalisés.

class tkinter.filedialog.**FileDialog**(*master*, *title=None*)

Crée une boîte de dialogue basique de sélection de fichiers.

cancel_command(*event=None*)

Déclenche la fermeture de la fenêtre de dialogue.

dirs_double_event(*event*)

Gestionnaire d'événements pour l'événement de double-clic sur le répertoire.

dirs_select_event(*event*)

Gestionnaire d'événements pour l'événement de clic sur le répertoire.

files_double_event(*event*)

Gestionnaire d'événements pour l'événement de double-clic sur le fichier.

files_select_event(*event*)

Gestionnaire d'événements pour un événement en un seul clic sur le fichier.

filter_command(*event=None*)

Filtre les fichiers par répertoire.

get_filter()

Récupère le filtre de fichiers actuellement utilisé.

get_selection()

Récupère l'élément actuellement sélectionné.

go(*dir_or_file=os.curdir*, *pattern='*'*, *default=''*, *key=None*)

Affiche la boîte de dialogue et lance la boucle d'événements.

ok_event(*event*)

Quitte la boîte de dialogue en renvoyant la sélection actuelle.

quit(*how=None*)

Sort de la boîte de dialogue en renvoyant le nom du fichier, le cas échéant.

set_filter(*dir*, *pat*)

Définit le filtre de fichiers.

set_selection(*file*)

Met à jour la sélection de fichiers actuelle vers *file*.

class tkinter.filedialog.**LoadFileDialog**(*master*, *title=None*)

Sous-classe de *FileDialog* qui crée une fenêtre de dialogue pour sélectionner un fichier existant.

ok_command()

Teste qu'un fichier est fourni et que la sélection indique un fichier déjà existant.

class tkinter.filedialog.**SaveFileDialog**(*master*, *title=None*)

Sous-classe de *FileDialog* qui crée une fenêtre de dialogue pour sélectionner un fichier de destination.

ok_command()

Teste si la sélection pointe vers un fichier valide qui n'est pas un répertoire. Une confirmation est requise si un fichier déjà existant est sélectionné.

26.4.3 `tkinter.commondialog` – Modèles de fenêtre de dialogue

Code source : Lib/tkinter/commondialog.py

Le module `tkinter.commondialog` fournit la classe `Dialog` qui est la classe mère pour les dialogues définis dans d'autres modules.

class `tkinter.commondialog.`**`Dialog`**(*master=None*, ***options*)

show(*color=None*, ***options*)

Affiche la fenêtre de dialogue.

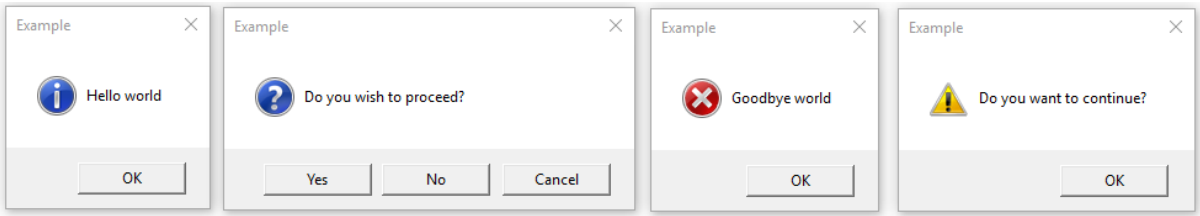
Voir aussi

Modules `tkinter.messagebox`, tut-files

26.5 `tkinter.messagebox` --- Tkinter message prompts

Code source : Lib/tkinter/messagebox.py

The `tkinter.messagebox` module provides a template base class as well as a variety of convenience methods for commonly used configurations. The message boxes are modal and will return a subset of (`True`, `False`, `None`, `OK`, `CANCEL`, `YES`, `NO`) based on the user's selection. Common message box styles and layouts include but are not limited to :



class `tkinter.messagebox.`**`Message`**(*master=None*, ***options*)

Create a message window with an application-specified message, an icon and a set of buttons. Each of the buttons in the message window is identified by a unique symbolic name (see the *type* options).

The following options are supported :

command
Specifies the function to invoke when the user closes the dialog. The name of the button clicked by the user to close the dialog is passed as argument. This is only available on macOS.

default
Gives the *symbolic name* of the default button for this message window (`OK`, `CANCEL`, and so on). If this option is not specified, the first button in the dialog will be made the default.

detail
Specifies an auxiliary message to the main message given by the *message* option. The message detail will be presented beneath the main message and, where supported by the OS, in a less emphasized font than the main message.

icon
Specifies an *icon* to display. If this option is not specified, then the `INFO` icon will be displayed.

message
Specifies the message to display in this message box. The default value is an empty string.

parent
Makes the specified window the logical parent of the message box. The message box is displayed on top of its parent window.

title
Specifies a string to display as the title of the message box. This option is ignored on macOS, where platform guidelines forbid the use of a title on this kind of dialog.

type
Arranges for a *predefined set of buttons* to be displayed.

show(***options*)

Display a message window and wait for the user to select one of the buttons. Then return the symbolic name of the selected button. Keyword arguments can override options specified in the constructor.

Boîte de message d'information

tkinter.messagebox.**showinfo**(*title=None*, *message=None*, ***options*)

Creates and displays an information message box with the specified title and message.

Boîtes de message d'avertissement

tkinter.messagebox.**showwarning**(*title=None*, *message=None*, ***options*)

Creates and displays a warning message box with the specified title and message.

tkinter.messagebox.**showerror**(*title=None*, *message=None*, ***options*)

Creates and displays an error message box with the specified title and message.

Boîtes de message posant une question

tkinter.messagebox.**askquestion**(*title=None*, *message=None*, ***, *type=YESNO*, ***options*)

Ask a question. By default shows buttons `YES` and `NO`. Returns the symbolic name of the selected button.

tkinter.messagebox.**askokcancel**(*title=None*, *message=None*, ***options*)

Ask if operation should proceed. Shows buttons `OK` and `CANCEL`. Returns `True` if the answer is ok and `False` otherwise.

tkinter.messagebox.**askretrycancel**(*title=None*, *message=None*, ***options*)

Ask if operation should be retried. Shows buttons `RETRY` and `CANCEL`. Return `True` if the answer is yes and `False` otherwise.

tkinter.messagebox.**askyesno**(*title=None*, *message=None*, ***options*)

Ask a question. Shows buttons `YES` and `NO`. Returns `True` if the answer is yes and `False` otherwise.

tkinter.messagebox.**askyesnocancel**(*title=None*, *message=None*, ***options*)

Ask a question. Shows buttons `YES`, `NO` and `CANCEL`. Return `True` if the answer is yes, `None` if cancelled, and `False` otherwise.

Symbolic names of buttons :

tkinter.messagebox.**ABORT = 'abort'**

tkinter.messagebox.**RETRY = 'retry'**

tkinter.messagebox.**IGNORE = 'ignore'**

tkinter.messagebox.**OK = 'ok'**

tkinter.messagebox.**CANCEL = 'cancel'**

tkinter.messagebox.**YES = 'yes'**

tkinter.messagebox.**NO = 'no'**

Predefined sets of buttons :

tkinter.messagebox.**ABORTRETRYIGNORE = 'abortretryignore'**

Displays three buttons whose symbolic names are `ABORT`, `RETRY` and `IGNORE`.

`tkinter.messagebox.OK = 'ok'`

Displays one button whose symbolic name is *OK*.

`tkinter.messagebox.OKCANCEL = 'okcancel'`

Displays two buttons whose symbolic names are *OK* and *CANCEL*.

`tkinter.messagebox.RETRYCANCEL = 'retrycancel'`

Displays two buttons whose symbolic names are *RETRY* and *CANCEL*.

`tkinter.messagebox.YESNO = 'yesno'`

Displays two buttons whose symbolic names are *YES* and *NO*.

`tkinter.messagebox.YESNOCANCEL = 'yesnocancel'`

Displays three buttons whose symbolic names are *YES*, *NO* and *CANCEL*.

Icon images :

`tkinter.messagebox.ERROR = 'error'`

`tkinter.messagebox.INFO = 'info'`

`tkinter.messagebox.QUESTION = 'question'`

`tkinter.messagebox.WARNING = 'warning'`

26.6 `tkinter.scrolledtext` --- Scrolled Text Widget

Code source : Lib/tkinter/scrolledtext.py

Le module *tkinter.scrolledtext* fournit une classe, de même nom, implémentant un simple gadget texte avec une barre de défilement verticale, configuré "pour faire ce qu'on attend de lui". Utiliser *ScrolledText* est beaucoup plus simple que configurer un gadget texte et une barre de défilement.

Le gadget texte et la barre de défilement sont regroupés dans une `Frame`, et les méthodes gestionnaires de géométrie `Grid` et `Pack` sont récupérées de l'objet `Frame`. L'objet *ScrolledText* a donc tous les attributs classiques pour la gestion de la géométrie.

Si un contrôle plus fin est nécessaire, les attributs suivants sont disponibles :

class `tkinter.scrolledtext.`**`ScrolledText`**(*master=None*, ***kw*)

`frame`

Le cadre (objet *Frame*) qui englobe le gadget texte et le gadget de la barre de défilement.

`vbar`

Le gadget de la barre de défilement.

26.7 `tkinter.dnd` --- Drag and drop support

Code source : Lib/tkinter/dnd.py

> **Note**
>
> Ceci est expérimental et devrait être obsolète lorsqu'il sera remplacé par le *Tk DND*.

Le module `tkinter.dnd` offre une prise en charge du glisser-déposer d'objets au sein d'une même application, dans la même fenêtre ou entre les fenêtres. Pour activer le glissement d'un objet, vous devez créer une liaison d'événements pour celui-ci qui démarre le processus de glisser-déposer. En règle générale, vous liez un événement *ButtonPress* à une fonction de rappel que vous écrivez (voir *Liaisons et événements*). La fonction doit appeler `dnd_start()`, où *source* est l'objet à faire glisser et *event* est l'événement qui a invoqué l'appel (l'argument de votre fonction de rappel).

La sélection d'un objet cible se produit comme suit :

1. Recherche descendante de la zone sous la souris pour le widget cible

— Le widget cible doit avoir un attribut appelable *dnd_accept*
— If *dnd_accept* is not present or returns `None`, search moves to parent widget
— If no target widget is found, then the target object is `None`

2. Appel à `<old_target>.dnd_leave(source, event)`
3. Appel à `<new_target>.dnd_enter(source, event)`
4. Appel à `<target>.dnd_commit(source, event)` pour notifier le déposer.
5. Appel à `<source>.dnd_end(target, event)` pour signaler la fin du glisser-déposer

class `tkinter.dnd.`**`DndHandler`**(*source*, *event*)

La classe *DndHandler* gère les événements de glisser-déposer qui suivent les événements *Motion* et *ButtonRelease* à la racine du widget d'événement.

`cancel`(*event=None*)

Annule le processus de glisser-déposer.

`finish`(*event*, *commit=0*)

Exécute les fonctions de fin de glisser-déposer.

`on_motion`(*event*)

Inspecte la zone sous la souris pour les objets cibles pendant que le glissement est effectué.

`on_release`(*event*)

Signale la fin de la traînée lorsque le modèle de relâchement est déclenché.

`tkinter.dnd.`**`dnd_start`**(*source*, *event*)

Fonction de fabrique pour le processus de glisser-déposer.

Voir aussi

Liaisons et événements

26.8 `tkinter.ttk` --- Tk themed widgets

Code source : Lib/tkinter/ttk.py

Le module `tkinter.ttk` donne accès à l'ensemble de widgets sur le thème *Tk*, introduit dans *Tk* 8.5. Il offre des avantages supplémentaires, notamment le rendu des polices avec anticrénelage sous *X11* et la transparence des fenêtres (nécessitant un gestionnaire de fenêtres de composition sur *X11*).

L'idée de base de `tkinter.ttk` est de séparer, dans la mesure du possible, le code implémentant le comportement d'un widget du code implémentant son apparence.

Voir aussi

Tk Widget Styling Support
Document présentant la prise en charge des thèmes pour *Tk*

26.8.1 Utilisation de *Ttk*

Pour commencer à utiliser *Ttk*, importez son module :

```
from tkinter import ttk
```

Pour remplacer les widgets *Tk* de base, l'importation doit suivre l'importation *Tk* :

```
from tkinter import *
from tkinter.ttk import *
```

Ce code implique que plusieurs widgets *tkinter.ttk* (`Button`, `Checkbutton`, `Entry`, `Frame`, `Label`, `LabelFrame`, `Menubutton`, `PanedWindow`, `Radiobutton`, `Scale` et `Scrollbar`) remplacent automatiquement les widgets *Tk*.

L'avantage immédiat est d'utiliser les nouveaux widgets qui donnent une meilleure apparence sur toutes les plateformes ; cependant, les widgets de remplacement ne sont pas complètement compatibles. La principale différence est que les options de widget telles que `"fg"`, `"bg"` et d'autres liées au style des widgets ne sont plus présentes dans les widgets *Ttk*. Utilisez plutôt la classe `ttk.Style` pour des effets de style améliorés.

Voir aussi

Converting existing applications to use Tile widgets
Monographie (utilisant la terminologie *Tcl*) sur les différences généralement rencontrées lors de la modification d'applications pour utiliser les nouveaux widgets.

26.8.2 Widgets *Ttk*

Ttk est livré avec 18 widgets, dont douze existaient déjà dans *tkinter* : `Button`, `Checkbutton`, `Entry`, `Frame`, `Label`, `LabelFrame`, `Menubutton`, `PanedWindow`, `Radiobutton`, `Scale`, `Scrollbar` et *Spinbox*. Les six autres sont nouveaux : *Combobox*, *Notebook*, *Progressbar*, `Separator`, `Sizegrip` et *Treeview*. Et tous sont des sous-classes de *Widget*.

L'utilisation des widgets *Ttk* donne à l'application une apparence et une convivialité améliorées. Comme indiqué ci-dessus, il existe des différences dans la façon dont le style est codé.

Code *Tk* :

```
l1 = tkinter.Label(text="Test", fg="black", bg="white")
l2 = tkinter.Label(text="Test", fg="black", bg="white")
```

Code *Ttk* :

```
style = ttk.Style()
style.configure("BW.TLabel", foreground="black", background="white")

l1 = ttk.Label(text="Test", style="BW.TLabel")
l2 = ttk.Label(text="Test", style="BW.TLabel")
```

Pour plus d'informations sur *TtkStyling*, consultez la documentation de la classe *Style*.

26.8.3 Widget

`ttk.Widget` définit les options et méthodes standard prises en charge par les widgets à thème *Tk* et n'est pas censé être directement instancié.

Options standards

All the `ttk` Widgets accept the following options :

Option	Description
class	Spécifie la classe de fenêtre. La classe est utilisée lors de l'interrogation de la base de données d'options pour les autres options de la fenêtre, pour déterminer les balises de liaison par défaut pour la fenêtre et pour sélectionner la disposition et le style par défaut du widget. Cette option est en lecture seule et ne peut être spécifiée qu'à la création de la fenêtre.
cursor	Spécifie le curseur de la souris à utiliser pour le widget. S'il est défini sur la chaîne vide (valeur par défaut), le curseur est hérité du widget parent.
takefocus	Détermine si la fenêtre accepte le focus pendant le parcours du clavier. 0, 1 ou une chaîne vide est renvoyée. Si 0 est renvoyé, cela signifie que la fenêtre doit être entièrement ignorée lors du parcours au clavier. Si 1, cela signifie que la fenêtre doit recevoir le focus d'entrée tant qu'elle est visible. Et une chaîne vide signifie que les scripts de parcours prennent la décision de donner ou non le focus à la fenêtre.
style	Peut être utilisé pour spécifier un style de widget personnalisé.

Options de widget avec barre de défilement

Les options suivantes sont prises en charge par les widgets contrôlés par une barre de défilement.

Option	Description
xscrollcommand	Utilisé pour communiquer avec les barres de défilement horizontal. Lorsque la vue dans la fenêtre du widget change, le widget génère une commande *Tcl* basée sur *scrollcommand*. Habituellement, cette option consiste en la méthode `Scrollbar.set()` d'une barre de défilement. Cela entraîne la mise à jour de la barre de défilement chaque fois que la vue dans la fenêtre change.
yscrollcommand	Utilisé pour communiquer avec les barres de défilement vertical. Pour plus d'informations, voir ci-dessus.

Options d'étiquette (*label*)

Les options suivantes sont prises en charge par les étiquettes (*labels*), les boutons et autres widgets de type bouton.

Option	Description
text	Spécifie une chaîne de texte à afficher dans le widget.
textvariable	Spécifie un nom dont la valeur est utilisée à la place de la ressource d'option de texte.
underline	S'il est défini, spécifie l'indice (commençant à 0) d'un caractère à souligner dans la chaîne de texte. Le caractère de soulignement est utilisé pour l'activation mnémonique.
image	Spécifie une image à afficher. C'est une liste de un ou plusieurs éléments. Le premier élément est le nom de l'image par défaut. Le reste de la liste est une séquence de paires *statespec-valeur* telles que définies par `Style.map()`, spécifiant différentes images à utiliser lorsque le widget est dans un état particulier ou une combinaison d'états. Toutes les images de la liste doivent avoir la même taille.
compound	Spécifie comment afficher l'image par rapport au texte, dans le cas où les options de texte et d'images sont présentes. Les valeurs valides sont : — *text* : afficher uniquement le texte — *image* : afficher uniquement l'image — *top*, *bottom*, *left*, *right* : afficher l'image au-dessus, en dessous, à gauche ou à droite du texte, respectivement. — *none* : la valeur par défaut. Afficher l'image si présente, sinon le texte.
width	Si supérieur à zéro, spécifie la quantité d'espace, en largeurs de caractères, à allouer à l'étiquette de texte, si inférieur à zéro, spécifie une largeur minimale. Si zéro ou non spécifié, la largeur naturelle de l'étiquette de texte est utilisée.

Options de compatibilité

Option	Description
state	Peut être défini sur `"normal"` ou `"disabled"` pour contrôler le bit d'état d'activation. Il s'agit d'une option en écriture seule : sa définition modifie l'état du widget, mais la méthode `Widget.state()` n'affecte pas cette option.

États des widgets

L'état du widget est une combinaison d'indicateurs d'état indépendants.

Option	Description
active	Le curseur de la souris est sur le widget et appuyer sur un bouton de la souris provoquera une action
disabled	Le widget est désactivé sous le contrôle du programme
focus	Le widget a le focus du clavier
pressed	Le widget est en train d'être pressé
selected	`"On"`, `"true"` ou `"current"` pour des choses comme les boutons de contrôle et les boutons radio
background	Windows et Mac ont une notion de fenêtre « active » ou de premier plan. L'état *background* est défini pour les widgets dans une fenêtre en arrière-plan et effacé pour ceux de la fenêtre au premier plan
readonly	Le widget ne doit pas permettre la modification par l'utilisateur
alternate	Un format d'affichage alternatif spécifique au widget
invalid	La valeur du widget est invalide

Une spécification d'état est une séquence de noms d'état, éventuellement précédée d'un point d'exclamation indiquant que le bit est désactivé.

ttk.Widget

Outre les méthodes décrites ci-dessous, la `ttk.Widget` prend en charge les méthodes `tkinter.Widget.cget()` et `tkinter.Widget.configure()`.

class `tkinter.ttk.`**`Widget`**

identify(*x*, *y*)

Renvoie le nom de l'élément à la position *x y*, ou la chaîne vide si le point ne se trouve dans aucun élément.

x et *y* sont des coordonnées en pixels relatives au widget.

instate(*statespec*, *callback=None*, **args*, ***kw*)

Teste l'état du widget. Si un rappel n'est pas spécifié, renvoie `True` si l'état du widget correspond à *statespec* et `False` sinon. Si le rappel est spécifié, il est appelé avec des arguments si l'état du widget correspond à *statespec*.

state(*statespec=None*)

Modifie ou demande l'état du widget. Si *statespec* est spécifié, définit l'état du widget en fonction de celui-ci et renvoie un nouveau *statespec* indiquant quels drapeaux ont été modifiés. Si *statespec* n'est pas spécifié, renvoie les indicateurs d'état actuellement activés.

statespec est généralement une liste ou un *n*-uplet.

26.8.4 Combobox

Le widget `ttk.Combobox` combine un champ de texte avec une liste déroulante de valeurs. Ce widget est une sous-classe de `Entry`.

En plus des méthodes héritées de *`Widget`* : `Widget.cget()`, `Widget.configure()`, *`Widget.identify()`*, *`Widget.instate()`* et *`Widget.state()`*, et les suivants hérités de `Entry` : `Entry.bbox()`, `Entry.delete()`, `Entry.icursor()`, `Entry.index()`, `Entry.insert()`, `Entry.selection()`, `Entry.xview()`, il a quelques autres méthodes, décrites dans `ttk.Combobox`.

Options

Ce widget accepte les options spécifiques suivantes :

Option	Description
exportationsselection	Valeur booléenne. Si elle est définie, la sélection du widget est liée à la sélection du gestionnaire de fenêtres (qui peut être renvoyée en appelant *Misc.selection_get*, par exemple).
justify	Spécifie comment le texte est aligné dans le widget. Les valeurs possibles sont `"left"`, `"center"` ou `"right"`.
height	Spécifie la hauteur de la liste déroulante, en lignes.
postcommand	Script (éventuellement enregistré avec *Misc.register*) qui est appelé immédiatement avant d'afficher les valeurs. Il peut spécifier les valeurs à afficher.
state	Les valeurs sont `"normal"`, `"readonly"` ou `"disabled"`. Dans l'état *readonly*, la valeur ne peut pas être modifiée directement et l'utilisateur ne peut que sélectionner les valeurs dans la liste déroulante. Dans l'état *normal*, le champ texte est directement éditable. Dans l'état *disabled*, aucune interaction n'est possible.
textvariable	Spécifie un nom dont la valeur est liée à la valeur du widget. Chaque fois que la valeur associée à ce nom change, la valeur du widget est mise à jour, et vice versa. Voir `tkinter.StringVar`.
values	Spécifie la liste de valeurs à afficher dans la liste déroulante.
width	Spécifie une valeur entière indiquant la largeur souhaitée de la fenêtre d'entrée, en caractères de taille moyenne de la police du widget.

Événements virtuels

Les widgets *combobox* génèrent un événement virtuel **«ComboboxSelected»** lorsque l'utilisateur sélectionne un élément dans la liste de valeurs.

ttk.Combobox

class tkinter.ttk.**Combobox**

current(*newindex=None*)

Si *newindex* est spécifié, définit la valeur de la *combobox* à la position de l'élément *newindex*. Sinon, renvoie l'indice de la valeur courante ou −1 si la valeur courante n'est pas dans la liste des valeurs.

get()

Renvoie la valeur actuelle de la *combobox*.

set(*value*)

Définit la valeur de la *combobox* sur *value*.

26.8.5 Spinbox

Le widget `ttk.Spinbox` est un `ttk.Entry` amélioré avec des flèches d'incrémentation et de décrémentation. Il peut être utilisé pour des nombres ou des listes de valeurs de chaîne. Ce widget est une sous-classe de `Entry`.

En plus des méthodes héritées de *`Widget`* : `Widget.cget()`, `Widget.configure()`, *`Widget.identify()`*, *`Widget.instate()`* et *`Widget.state()`*, et les suivantes héritées de `Entry` : `Entry.bbox()`, `Entry.delete()`, `Entry.icursor()`, `Entry.index()`, `Entry.insert()`, `Entry.xview()`, il a quelques autres méthodes, décrites dans `ttk.Spinbox`.

Options

Ce widget accepte les options spécifiques suivantes :

Option	Description
from	Valeur flottante. Si elle est définie, il s'agit de la valeur minimale à laquelle le bouton de décrémentation décrémentera. Doit être orthographié comme `from_` lorsqu'il est utilisé comme argument, car `from` est un mot-clé Python.
to	Valeur flottante. Si elle est définie, il s'agit de la valeur maximale du bouton d'incrémentation.
increment	Valeur flottante. Spécifie la quantité dont les boutons d'incrémentation/décrémentation modifient la valeur. La valeur par défaut est 1.0.
values	Séquence de valeurs de chaîne ou flottantes. S'ils sont spécifiés, les boutons d'incrémentation/décrémentation font défiler les éléments dans cette séquence plutôt que d'incrémenter ou de décrémenter les nombres.
wrap	Valeur booléenne. Si `True`, les boutons d'incrémentation et de décrémentation passent de la valeur `to` à la valeur `from` ou de la valeur `from` à la valeur `to`, respectivement.
format	Valeur sous forme de chaîne. Cela spécifie le format des nombres définis par les boutons d'incrémentation/décrémentation. Il doit être sous la forme `"%W.Pf"`, où W est la largeur de la valeur, P est la précision et `'%'` et `'f'` sont littéraux.
command	appelable Python. Est appelé sans argument chaque fois que l'un des boutons d'incrémentation ou de décrémentation est enfoncé.

Événements virtuels

Le widget *spinbox* génère un événement virtuel **«Increment»** lorsque l'utilisateur appuie sur <Up>, et un événement virtuel **«Decrement»** lorsque l'utilisateur appuie sur <Down>.

ttk.Spinbox

class tkinter.ttk.**Spinbox**

get()

Renvoie la valeur actuelle du *spinbox*.

set(*value*)

Définit la valeur du *spinbox* sur *value*.

26.8.6 Carnet de notes (*notebook*)

Le widget *Ttk Notebook* gère une collection de fenêtres et n'en affiche qu'une seule à la fois. Chaque fenêtre enfant est associée à un onglet, que l'utilisateur peut sélectionner pour changer la fenêtre actuellement affichée.

Options

Ce widget accepte les options spécifiques suivantes :

Option	Description
height	S'il est présent et supérieur à zéro, spécifie la hauteur souhaitée de la zone du volet (sans compter l'ajustement interne ni les onglets). Sinon, la hauteur maximale de tous les volets est utilisée.
padding	Spécifie la quantité d'espace supplémentaire à ajouter autour de l'extérieur du bloc-notes. L'ajustement est défini par une liste jusqu'à quatre spécifications de longueur à gauche en haut à droite en bas. Si moins de quatre éléments sont spécifiés, *bottom* vaut par défaut *top*, *right* vaut par défaut *left* et *top* vaut par défaut *left*.
width	S'il est présent et supérieur à zéro, spécifiez la largeur souhaitée de la zone du volet (hors ajustement interne). Sinon, la largeur maximale de tous les volets est utilisée.

Options d'onglet

Il existe également des options spécifiques pour les onglets :

Option	Description
state	Soit `"normal"`, `"disabled"` ou `"hidden"`. Si *disabled*, l'onglet n'est pas sélectionnable. S'il est *hidden*, l'onglet n'est pas affiché.
sticky	Spécifie comment la fenêtre enfant est positionnée dans la zone du volet. La valeur est une chaîne contenant zéro ou plusieurs des caractères "n", "s", "e" ou "w". Chaque lettre fait référence à un côté (nord, sud, est ou ouest) auquel la fenêtre enfant colle, selon le gestionnaire de géométrie `grid()`.
padding	Spécifie la quantité d'espace supplémentaire à ajouter entre le bloc-notes et ce volet. La syntaxe est la même que pour l'option d'ajustement utilisée par ce widget.
text	Spécifie un texte à afficher dans l'onglet.
image	Spécifie une image à afficher dans l'onglet. Voir l'option *image* décrite dans `Widget`.
compound	Spécifie comment afficher l'image par rapport au texte, dans le cas où les deux options texte et image sont présentes. Voir *Label Options* pour les valeurs autorisées.
underline	Spécifie l'indice (en commençant à 0) d'un caractère à souligner dans la chaîne de texte. Le caractère souligné est utilisé pour l'activation mnémonique si `Notebook.enable_traversal()` est appelé.

Identifiants d'onglet

Le *tab_id* présent dans plusieurs méthodes de `ttk.Notebook` peut prendre l'une des formes suivantes :

- Un entier compris entre zéro et le nombre d'onglets
- Le nom d'une fenêtre fille
- Une spécification de position de la forme "@x,y", qui identifie l'onglet
- La chaîne littérale `"current"`, qui identifie l'onglet actuellement sélectionné

— La chaîne littérale `"end"`, qui renvoie le nombre d'onglets (valable uniquement pour `Notebook.index()`)

Événements virtuels

Ce widget génère un événement virtuel **«NotebookTabChanged»** après la sélection d'un nouvel onglet.

ttk.Notebook

class `tkinter.ttk.`**Notebook**

add(*child*, ***kw*)

Ajoute un nouvel onglet au bloc-notes.

Si la fenêtre est actuellement gérée par le *notebook* mais masquée, elle est restaurée à sa position précédente.

Voir *Options d'onglet* pour la liste des options disponibles.

forget(*tab_id*)

Supprime l'onglet spécifié par *tab_id*, et ne gère plus la fenêtre associée.

hide(*tab_id*)

Masque l'onglet spécifié par *tab_id*.

L'onglet ne s'affiche pas, mais la fenêtre associée reste gérée par le *notebook* et sa configuration mémorisée. Les onglets cachés peuvent être restaurés avec la commande `add()`.

identify(*x*, *y*)

Renvoie le nom de l'élément *tab* à la position *x*, *y*, ou la chaîne vide s'il n'y en a pas.

index(*tab_id*)

Renvoie l'indice de l'onglet spécifié par *tab_id*, ou le nombre total d'onglets si *tab_id* est la chaîne `"end"`.

insert(*pos*, *child*, ***kw*)

Insère un volet à la position spécifiée.

pos est soit la chaîne `"end"`, un indice entier ou le nom d'un enfant géré. Si *child* est déjà géré par le *notebook*, le déplace vers la position spécifiée.

Voir *Options d'onglet* pour la liste des options disponibles.

select(*tab_id=None*)

Sélectionne le *tab_id* spécifié.

La fenêtre enfant associée est affichée, et la fenêtre précédemment sélectionnée (si différente) est masquée. Si *tab_id* est omis, renvoie le nom du widget du volet actuellement sélectionné.

tab(*tab_id*, *option=None*, ***kw*)

Interroge ou modifie les options du *tab_id* spécifique.

Si *kw* n'est pas donné, renvoie un dictionnaire des valeurs des options de l'onglet. Si *option* est spécifié, renvoie la valeur de cette *option*. Sinon, définit les options sur les valeurs correspondantes.

tabs()

Renvoie une liste des fenêtres gérées par le *notebook*.

enable_traversal()

Active la traversée du clavier pour une fenêtre de niveau supérieur contenant ce bloc-notes.

Cela étend les liaisons pour la fenêtre de niveau supérieur contenant le bloc-notes comme suit :
— `Control-Tab` : sélectionne l'onglet suivant celui actuellement sélectionné.
— `Shift-Control-Tab` : sélectionne l'onglet précédant celui actuellement sélectionné.
— `Alt-K` : où *K* est le caractère mnémonique (souligné) de n'importe quel onglet, sélectionne cet onglet.
Plusieurs blocs-notes dans un seul niveau supérieur peuvent être activés pour la traversée, y compris les blocs-notes imbriqués. Cependant, la traversée du bloc-notes ne fonctionne correctement que si tous les volets ont le bloc-notes dans lequel ils se trouvent en tant que maître.

26.8.7 Barre de progression

Le widget `ttk.Progressbar` affiche l'état d'une opération de longue durée. Il peut fonctionner en deux modes : 1) le mode déterminé qui indique la quantité achevée par rapport à la quantité totale de travail à effectuer et 2) le mode indéterminé qui fournit un affichage animé pour informer l'utilisateur que le travail progresse.

Options

Ce widget accepte les options spécifiques suivantes :

Option	Description
orient	`"horizontal"` ou `"vertical"`. Spécifie l'orientation de la barre de progression.
length	Spécifie la longueur de l'axe long de la barre de progression (largeur si horizontale, hauteur si verticale).
mode	`"determinate"` ou `"indeterminate"`.
maximum	Nombre spécifiant la valeur maximale. La valeur par défaut est 100.
value	La valeur actuelle de la barre de progression. En mode *determinate*, cela représente la quantité de travail accompli. En mode *indeterminate*, il est interprété comme modulo *maximum* ; c'est-à-dire que la barre de progression effectue un « cycle » lorsque sa valeur augmente de *maximum*.
variable	Nom lié à la valeur de l'option. Si spécifié, la valeur de la barre de progression est automatiquement fixée à la valeur de ce nom à chaque modification de ce dernier.
phase	Option en lecture seule. Le widget incrémente périodiquement la valeur de cette option chaque fois que sa valeur est supérieure à 0 et, en mode *déterminate*, inférieure au maximum. Cette option peut être utilisée par le thème actuel pour fournir des effets d'animation supplémentaires.

ttk.Progressbar

class `tkinter.ttk.`**`Progressbar`**

start(*interval=None*)

Commence le mode d'auto-incrémentation : planifie un événement de minuterie récurrent qui appelle `Progressbar.step()` toutes les *interval* millisecondes. S'il est omis, *interval* est par défaut de 50 millisecondes.

step(*amount=None*)

Incrémente la valeur de la barre de progression de *amount*.

amount est par défaut égal à `1.0` s'il est omis.

stop()

Arrête le mode d'auto-incrémentation : annule tout événement de minuterie récurrent initié par `Progressbar.start()` pour cette barre de progression.

26.8.8 Séparateur

Le widget `ttk.Separator` affiche une barre de séparation horizontale ou verticale.

Il n'a pas d'autres méthodes que celles héritées de `ttk.Widget`.

Options

Ce widget accepte l'option spécifique suivante :

Option	Description
orient	`"horizontal"` ou `"vertical"`. Spécifie l'orientation du séparateur.

26.8.9 Poignée de redimensionnement

Le widget `ttk.Sizegrip` (également connu sous le nom de *grow box*) permet à l'utilisateur de redimensionner la fenêtre de niveau supérieur en appuyant et en faisant glisser la poignée.

Ce widget n'a ni options spécifiques ni méthodes spécifiques, à part celles héritées de `ttk.Widget`.

Notes spécifiques à une plateforme

— Sur macOS, les fenêtres de niveau supérieur incluent automatiquement une poignée de redimensionnement intégrée par défaut. L'ajout d'un `Sizegrip` est sans danger, car la poignée intégrée masquera simplement le widget.

Bogues

— Si la position du contenant de niveau supérieur a été spécifiée par rapport à la droite ou au bas de l'écran (par exemple…), le widget `Sizegrip` ne redimensionne pas la fenêtre.
— Ce widget ne prend en charge que le redimensionnement `"southeast"`.

26.8.10 Arborescence

Le widget `ttk.Treeview` affiche une collection hiérarchique d'éléments. Chaque élément possède une étiquette textuelle, une image facultative et une liste facultative de valeurs de données. Les valeurs des données sont affichées dans des colonnes successives après l'étiquette de l'arbre.

L'ordre dans lequel les valeurs de données sont affichées peut être contrôlé en définissant l'option de widget `displaycolumns`. Le widget d'arborescence peut également afficher les en-têtes de colonne. Les colonnes sont accessibles par des numéros ou des noms symboliques indiqués dans l'option *columns* du widget. Voir *Identifiants de colonnes*.

Chaque élément est identifié par un nom unique. Le widget génère des identifiants d'éléments s'ils ne sont pas fournis par l'appelant. Il existe un élément racine distinct, nommé `{}`. L'élément racine lui-même n'est pas affiché ; ses enfants apparaissent au niveau supérieur de la hiérarchie.

Chaque élément possède également une liste de balises, qui peuvent être utilisées pour associer des liaisons d'événements à des éléments individuels et contrôler l'apparence de l'élément.

Le widget *Treeview* prend en charge le défilement horizontal et vertical, selon les options décrites dans *Scrollable Widget Options* et les méthodes `Treeview.xview()` et `Treeview.yview()`.

Options

Ce widget accepte les options spécifiques suivantes :

Option	Description
columns	Une liste d'identificateurs de colonne, spécifiant le nombre de colonnes et leurs noms.
displaycolumns	Une liste d'identificateurs de colonne (indices symboliques ou entiers) spécifiant quelles colonnes de données sont affichées et l'ordre dans lequel elles apparaissent, ou la chaîne `"#all"`.
height	Spécifie le nombre de lignes qui doivent être visibles. Remarque : la largeur demandée est déterminée à partir de la somme des largeurs de colonne.
padding	Spécifie le remplissage interne du widget. L'ajustement est décrit par une liste jusqu'à quatre spécifications de longueur.
selectmode	Contrôle la façon dont les liaisons de classe intégrées gèrent la sélection. Les valeurs possibles sont `"extended"`, `"browse"` ou `"none"`. S'il est défini sur *extended* (valeur par défaut), plusieurs éléments peuvent être sélectionnés. Si c'est *browse* un seul élément ne peut être sélectionné à la fois. Si c'est *none*, la sélection ne peut pas être modifiée. Notez que le code d'application et les liaisons de balises peuvent définir la sélection comme ils le souhaitent, quelle que soit la valeur de cette option.
show	Une liste contenant zéro ou plusieurs des valeurs suivantes, spécifiant les éléments de l'arborescence à afficher. — *tree* : affiche les étiquettes de l'arbre dans la colonne 0. — *headings* : affiche la ligne d'en-tête. La valeur par défaut est `"tree headings"`, c'est-à-dire afficher tous les éléments. **Remarque** : la colonne 0 fait toujours référence à la colonne de l'arborescence, même si `show="tree"` n'est pas spécifié.

Options d'éléments

Les options d'éléments suivantes peuvent être spécifiées pour les éléments dans les commandes *insert* et les widgets de type élément.

Option	Description
text	Libellé textuel à afficher pour l'élément.
image	Image *Tk*, affichée à gauche de l'étiquette.
values	Liste des valeurs associées à l'élément. Chaque élément doit avoir le même nombre de valeurs que les colonnes d'options du widget. S'il y a moins de valeurs que de colonnes, les valeurs restantes sont supposées vides. S'il y a plus de valeurs que de colonnes, les valeurs supplémentaires sont ignorées.
open	Valeur `True` ou `False` indiquant si les enfants de l'élément doivent être affichés ou masqués.
tags	Liste de balises associées à cet élément.

Options de balise

Les options suivantes peuvent être spécifiées sur les balises :

Option	Description
foreground	Spécifie la couleur de premier plan du texte.
background	Spécifie la couleur d'arrière-plan de la cellule ou de l'élément.
font	Spécifie la police à utiliser lors du dessin du texte.
image	Spécifie l'image de l'élément, au cas où l'option d'image de l'élément est vide.

Identifiants de colonnes

Les identifiants de colonnes prennent l'une des formes suivantes :

- Un nom symbolique de l'option de liste de colonnes.
- Un entier n, spécifiant la n^{ième} colonne de données.
- Une chaîne de la forme #n, où n est un entier, spécifiant la n^{ième} colonne d'affichage.

Notes :

- Les valeurs d'option de l'élément peuvent être affichées dans un ordre différent de celui dans lequel elles sont stockées.
- La colonne #0 fait toujours référence à la colonne de l'arborescence, même si `show="tree"` n'est pas spécifié.

Un numéro de colonne de données est un indice dans la liste des valeurs d'option d'un élément ; un numéro de colonne d'affichage est le numéro de colonne dans l'arborescence où les valeurs sont affichées. Les étiquettes d'arbre sont affichées dans la colonne #0. Si l'option *displaycolumns* n'est pas définie, la colonne de données n s'affiche dans la colonne #n+1. Encore une fois, **la colonne #0 fait toujours référence à la colonne de l'arborescence**.

Événements virtuels

Le widget *Treeview* génère les événements virtuels suivants.

Événement	Description
«TreeviewSelect»	Généré chaque fois que la sélection change.
«TreeviewOpen»	Généré juste avant de définir l'élément de focus sur `open=True`.
«TreeviewClose»	Généré juste après avoir défini l'élément de focus sur `open=False`.

Les méthodes `Treeview.focus()` et `Treeview.selection()` peuvent être utilisées pour déterminer le ou les éléments concernés.

ttk.Treeview

class tkinter.ttk.**Treeview**

bbox(*item*, *column=None*)

Renvoie la boîte englobante (relative à la fenêtre du widget *treeview*) de l'*item* spécifié sous la forme (x, y, largeur, hauteur).

Si *column* est spécifié, renvoie la boîte englobante de cette cellule. Si *item* n'est pas visible (c'est-à-dire s'il est un descendant d'un élément fermé ou se trouve hors écran en raison du défilement), renvoie une chaîne vide.

get_children(*item=None*)

Renvoie la liste des enfants appartenant à *item*.

Si *item* n'est pas spécifié, renvoie les enfants racine.

set_children(*item*, **newchildren*)

Remplace l'enfant de *item* par *newchildren*.

Les enfants présents dans *item* qui ne sont pas présents dans *newchildren* sont détachés de l'arbre. Aucun élément de *newchildren* ne peut être un ancêtre de *item*. Notez que ne pas spécifier *newchildren* entraîne le détachement des enfants de *item*.

column(*column*, *option=None*, ***kw*)

Interroge ou modifie les options pour la *column* spécifiée.

Si *kw* n'est pas donné, renvoie un dictionnaire des valeurs d'option de colonne. Si *option* est spécifié, la valeur de cette *option* est renvoyée. Sinon, définit les options sur les valeurs correspondantes.

Les options/valeurs valides sont :

id

Renvoie le nom de la colonne. Il s'agit d'une option en lecture seule.

***anchor* : One of the standard Tk anchor values.**
Spécifie comment le texte de cette colonne doit être aligné par rapport à la cellule.

***minwidth* : width**
La largeur minimale de la colonne en pixels. Le widget *Treeview* ne rend pas la colonne plus petite que celle spécifiée par cette option lorsque le widget est redimensionné ou que l'utilisateur fait glisser une colonne.

***stretch* : `True`/`False`**
Spécifie si la largeur de la colonne doit être ajustée lorsque le widget est redimensionné.

***width* : width**
La largeur de la colonne en pixels.

Pour configurer la colonne d'arborescence, appelez-la avec `column = "#0"`

`delete`(**items*)

Supprime tous les *items* spécifiés et tous leurs descendants.

L'élément racine ne peut pas être supprimé.

`detach`(**items*)

Dissocie tous les *items* spécifiés de l'arborescence.

Les éléments et tous leurs descendants sont toujours présents et peuvent être réinsérés à un autre point de l'arborescence, mais ne seront pas affichés.

L'élément racine ne peut pas être détaché.

`exists`(*item*)

Renvoie `True` si l'*item* spécifié est présent dans l'arborescence.

`focus`(*item=None*)

Si *item* est spécifié, définit l'élément de focus sur *item*. Sinon, renvoie l'élément de focus actuel, ou `''` s'il n'y en a pas.

`heading`(*column*, *option=None*, ***kw*)

Interroge ou modifie les options d'en-tête pour la *column* spécifiée.

Si *kw* n'est pas donné, renvoie un dictionnaire des valeurs d'option d'en-tête. Si *option* est spécifié, la valeur de cette *option* est renvoyée. Sinon, définit les options sur les valeurs correspondantes.

Les options/valeurs valides sont :

***text* : text**
Texte à afficher dans l'en-tête de colonne.

***image* : imageName**
Spécifie une image à afficher à droite de l'en-tête de colonne.

***anchor* : anchor**
Spécifie comment le texte du titre doit être aligné. Une des valeurs d'ancrage *Tk* standard.

***command* : callback**
Un rappel à invoquer lorsque l'étiquette d'en-tête est pressée.

Pour configurer l'en-tête de colonne de l'arborescence, appelez-la avec `column = "#0"`.

`identify`(*component*, *x*, *y*)

Renvoie une description du *component* spécifié sous le point donné par *x* et *y*, ou la chaîne vide si aucun *component* de ce type n'est présent à cette position.

`identify_row`(*y*)

Renvoie l'ID d'élément de l'élément à la position *y*.

`identify_column`(*x*)

Renvoie l'identificateur de colonne de données de la cellule à la position *x*.

La colonne arborescence possède l'ID #0.

identify_region(*x*, *y*)

Renvoie l'un des éléments suivants :

region	signification
heading	Zone d'en-tête de l'arbre.
separator	Espace entre deux en-têtes de colonnes.
tree (arbre)	Une zone de l'arbre.
cell	Une cellule de données.

Disponibilité : *Tk* 8.6.

identify_element(*x*, *y*)

Renvoie l'élément à la position *x*, *y*.

Disponibilité : *Tk* 8.6.

index(*item*)

Renvoie l'indice (entier) de *item* dans la liste des enfants de son parent.

insert(*parent*, *index*, *iid=None*, ***kw*)

Crée un nouvel élément et renvoie l'identifiant de l'élément nouvellement créé.

parent est l'ID d'élément de l'élément parent ou la chaîne vide pour créer un nouvel élément de niveau supérieur. *index* est un entier, ou la valeur `"end"`, spécifiant où dans la liste des enfants du parent insérer le nouvel élément. Si *index* est inférieur ou égal à zéro, le nouveau nœud est inséré au début ; si *index* est supérieur ou égal au nombre actuel d'enfants, il est inséré à la fin. Si *iid* est spécifié, il est utilisé comme identifiant d'élément ; *iid* ne doit pas déjà exister dans l'arborescence. Sinon, un nouvel identifiant unique est généré.

See *Item Options* for the list of available options.

item(*item*, *option=None*, ***kw*)

Interroge ou modifie les options pour l'*item* spécifié.

Si aucune option n'est donnée, un dictionnaire avec des options/valeurs pour l'élément est renvoyé. Si *option* est spécifié, la valeur de cette option est renvoyée. Sinon, définit les options sur les valeurs correspondantes données par *kw*.

move(*item*, *parent*, *index*)

Déplace *item* vers la position *index* dans la liste des enfants de *parent*.

Il est interdit de déplacer un élément sous l'un de ses descendants. Si *index* est inférieur ou égal à zéro, *item* est déplacé au début ; s'il est supérieur ou égal au nombre d'enfants, il est déplacé à la fin. Si *item* a été détaché, il est rattaché.

next(*item*)

Renvoie l'identifiant du frère suivant de *item*, ou `''` si *item* est le dernier enfant de son parent.

parent(*item*)

Renvoie l'identifiant du parent de *item*, ou `''` si *item* est au niveau supérieur de la hiérarchie.

prev(*item*)

Renvoie l'identifiant du frère précédent de *item*, ou `''` si *item* est le premier enfant de son parent.

reattach(*item*, *parent*, *index*)

Alias pour `Treeview.move()`.

see(*item*)

Fait en sorte que *item* soit visible.

Définit l'option d'ouverture de tous les ancêtres de *item* sur `True`, et fait défiler le widget si nécessaire afin que *item* soit dans la partie visible de l'arborescence.

selection()

Renvoie un *n*-uplet d'éléments sélectionnés.

Modifié dans la version 3.8 : `selection()` ne prend plus d'arguments. Pour modifier l'état de sélection, utilisez les méthodes de sélection suivantes.

selection_set(**items*)

items devient la nouvelle sélection.

Modifié dans la version 3.6 : *items* peut être passé en tant qu'arguments séparés, pas seulement en tant que *n*-uplet.

selection_add(**items*)

Ajoute des *items* à la sélection.

Modifié dans la version 3.6 : *items* peut être passé en tant qu'arguments séparés, pas seulement en tant que *n*-uplet.

selection_remove(**items*)

Supprime les *items* de la sélection.

Modifié dans la version 3.6 : *items* peut être passé en tant qu'arguments séparés, pas seulement en tant que *n*-uplet.

selection_toggle(**items*)

Bascule l'état de sélection de chaque élément de *items*.

Modifié dans la version 3.6 : *items* peut être passé en tant qu'arguments séparés, pas seulement en tant que *n*-uplet.

set(*item*, *column=None*, *value=None*)

Avec un argument, renvoie un dictionnaire de paires colonne/valeur pour l'*item* spécifié. Avec deux arguments, renvoie la valeur actuelle de la *column* spécifiée. Avec trois arguments, définit la valeur de la *column* donnée dans l'*item* donné à la *value* spécifiée.

tag_bind(*tagname*, *sequence=None*, *callback=None*)

Lie le rappel *callback* à l'événement donné *sequence* pour la balise *tagname*. Lorsqu'un événement est envoyé à un élément, les rappels pour chacune des options de balises de l'élément sont appelés.

tag_configure(*tagname*, *option=None*, ***kw*)

Interroge ou modifie les options pour le *tagname* spécifié.

Si *kw* n'est pas donné, renvoie un dictionnaire des paramètres d'option pour *tagname*. Si *option* est spécifié, renvoie la valeur de cette *option* pour le *tagname* spécifié. Sinon, définit les options sur les valeurs correspondantes pour le *tagname* donné.

tag_has(*tagname*, *item=None*)

Si *item* est spécifié, renvoie 1 ou 0 selon que *item* spécifié a le *tagname* donné. Sinon, renvoie une liste de tous les éléments qui ont la balise spécifiée.

Disponibilité : *Tk* 8.6

xview(**args*)

Interroge ou modifie la position horizontale de l'arborescence.

yview(**args*)

Interroge ou modifie la position verticale de l'arborescence.

26.8.11 Style Ttk

Chaque widget dans `ttk` se voit attribuer un style, qui spécifie l'ensemble d'éléments composant le widget et la façon dont ils sont disposés, ainsi que les paramètres dynamiques et par défaut pour les options d'élément. Par défaut, le nom du style est le même que le nom de la classe du widget, mais il peut être remplacé par l'option de style du widget. Si vous ne connaissez pas le nom de classe d'un widget, utilisez la méthode `Misc.winfo_class()` (*somewidget.winfo_class()*).

Voir aussi

Tcl'2004 conference presentation
Ce document explique le fonctionnement du moteur de thème

class tkinter.ttk.**Style**

Cette classe est utilisée pour manipuler la base de données de style.

configure(*style*, *query_opt=None*, ***kw*)

Interroge ou définit la valeur par défaut des options spécifiées dans *style*.

Chaque clé dans *kw* est une option et chaque valeur est une chaîne identifiant la valeur de cette option.

Par exemple, pour changer chaque bouton par défaut en un bouton plat avec un ajustement et une couleur d'arrière-plan différente :

```
from tkinter import ttk
import tkinter

root = tkinter.Tk()

ttk.Style().configure("TButton", padding=6, relief="flat",
   background="#ccc")

btn = ttk.Button(text="Sample")
btn.pack()

root.mainloop()
```

map(*style*, *query_opt=None*, ***kw*)

Interroge ou définit les valeurs dynamiques des options spécifiées dans *style*.

Chaque clé dans *kw* est une option et chaque valeur doit être une liste ou un *n*-uplet (généralement) contenant des spécifications d'état regroupées dans des *n*-uplets, des listes ou une autre préférence. Un *statespec* est un composé d'un ou plusieurs états, puis d'une valeur.

Un exemple peut le rendre plus compréhensible :

```
import tkinter
from tkinter import ttk

root = tkinter.Tk()

style = ttk.Style()
style.map("C.TButton",
    foreground=[('pressed', 'red'), ('active', 'blue')],
    background=[('pressed', '!disabled', 'black'), ('active', 'white')]
    )

colored_btn = ttk.Button(text="Test", style="C.TButton").pack()

root.mainloop()
```

Notez que l'ordre des séquences (états, valeur) pour une option est important, si l'ordre est changé en `[('active', 'blue'), ('pressed', 'red')]` dans l'option de premier plan, par exemple, le résultat est un premier plan bleu lorsque le widget est dans les états actif ou pressé.

lookup(*style*, *option*, *state=None*, *default=None*)

Renvoie la valeur spécifiée pour *option* dans *style*.

Si *state* est spécifié, on s'attend à ce qu'il s'agisse d'une séquence d'un ou plusieurs états. Si l'argument *default* est défini, il est utilisé comme valeur de secours au cas où aucune spécification d'option n'est trouvée.

Pour vérifier quelle police un bouton utilise par défaut :

```
from tkinter import ttk

print(ttk.Style().lookup("TButton", "font"))
```

layout (*style*, *layoutspec=None*)

Définit la disposition du widget pour un *style* donné. Si *layoutspec* est omis, renvoie la spécification de mise en page pour le style donné.

layoutspec, si spécifié, doit être une liste ou un autre type de séquence (à l'exception des chaînes), où chaque élément doit être un *n*-uplet et le premier élément est le nom de la mise en page et le deuxième élément doit avoir le format décrit dans *Layouts*.

Pour comprendre le format, consultez l'exemple suivant (il n'est pas destiné à faire quoi que ce soit d'utile) :

```
from tkinter import ttk
import tkinter

root = tkinter.Tk()

style = ttk.Style()
style.layout("TMenubutton", [
    ("Menubutton.background", None),
    ("Menubutton.button", {"children":
        [("Menubutton.focus", {"children":
            [("Menubutton.padding", {"children":
                [("Menubutton.label", {"side": "left", "expand": 1})]
            })]
        })]
    }),
])

mbtn = ttk.Menubutton(text='Text')
mbtn.pack()
root.mainloop()
```

element_create (*elementname*, *etype*, **args*, ***kw*)

Create a new element in the current theme, of the given *etype* which is expected to be either "image", "from" or "vsapi". The latter is only available in Tk 8.6 on Windows.

Si *image* est utilisé, *args* doit contenir le nom de l'image par défaut suivi de paires *statespec* / *value* (il s'agit de l'*imagespec*), et *kw* peut avoir les options suivantes :

***border*=remplissage**
padding est une liste de quatre entiers maximum, spécifiant respectivement les bordures gauche, supérieure, droite et inférieure.

***height*=hauteur**
Spécifie une hauteur minimale pour l'élément. Si elle est inférieure à zéro, la hauteur de l'image de base est utilisée par défaut.

***padding*= ajustement**
Spécifie l'ajustement intérieur de l'élément. La valeur par défaut est la valeur de *border* si elle n'est pas spécifiée.

sticky*= *spec
Spécifie comment l'image est placée dans la partie de l'espace attribué finale. *spec* contient zéro ou plusieurs caractères "n", "s", "w" ou "e".

***width*=largeur**
Spécifie une largeur minimale pour l'élément. Si inférieur à zéro, la largeur de l'image de base est utilisée par défaut.

Exemple :

```
img1 = tkinter.PhotoImage(master=root, file='button.png')
img1 = tkinter.PhotoImage(master=root, file='button-pressed.png')
img1 = tkinter.PhotoImage(master=root, file='button-active.png')
style = ttk.Style(root)
style.element_create('Button.button', 'image',
                     img1, ('pressed', img2), ('active', img3),
                     border=(2, 4), sticky='we')
```

Si *from* est utilisé comme valeur de *etype*, `element_create()` clone un élément existant. *args* doit contenir un nom de thème, à partir duquel l'élément est cloné, et éventuellement un élément à cloner. Si cet élément qu'il faut cloner n'est pas spécifié, un élément vide est utilisé. *kw* est ignoré.

Exemple :

```
style = ttk.Style(root)
style.element_create('plain.background', 'from', 'default')
```

If "vsapi" is used as the value of *etype*, `element_create()` will create a new element in the current theme whose visual appearance is drawn using the Microsoft Visual Styles API which is responsible for the themed styles on Windows XP and Vista. *args* is expected to contain the Visual Styles class and part as given in the Microsoft documentation followed by an optional sequence of tuples of ttk states and the corresponding Visual Styles API state value. *kw* may have the following options :

***padding*= ajustement**
Specify the element's interior padding. *padding* is a list of up to four integers specifying the left, top, right and bottom padding quantities respectively. If fewer than four elements are specified, bottom defaults to top, right defaults to left, and top defaults to left. In other words, a list of three numbers specify the left, vertical, and right padding; a list of two numbers specify the horizontal and the vertical padding; a single number specifies the same padding all the way around the widget. This option may not be mixed with any other options.

margins=padding
Specifies the elements exterior padding. *padding* is a list of up to four integers specifying the left, top, right and bottom padding quantities respectively. This option may not be mixed with any other options.

***width*=largeur**
Specifies the width for the element. If this option is set then the Visual Styles API will not be queried for the recommended size or the part. If this option is set then *height* should also be set. The *width* and *height* options cannot be mixed with the *padding* or *margins* options.

***height*=hauteur**
Specifies the height of the element. See the comments for *width*.

Exemple :

```
style = ttk.Style(root)
style.element_create('pin', 'vsapi', 'EXPLORERBAR', 3, [
                     ('pressed', '!selected', 3),
                     ('active', '!selected', 2),
                     ('pressed', 'selected', 6),
                     ('active', 'selected', 5),
                     ('selected', 4),
                     ('', 1)])
style.layout('Explorer.Pin',
             [('Explorer.Pin.pin', {'sticky': 'news'})])
pin = ttk.Checkbutton(style='Explorer.Pin')
pin.pack(expand=True, fill='both')
```

Modifié dans la version 3.13 : Added support of the "vsapi" element factory.

element_names()

Renvoie la liste des éléments définis dans le thème courant.

element_options(*elementname*)

Renvoie la liste des options de *elementname*.

theme_create(*themename*, *parent=None*, *settings=None*)

Crée un nouveau thème.

C'est une erreur si *themename* existe déjà. Si *parent* est spécifié, le nouveau thème hérite des styles, des éléments et des mises en page du thème parent. Si *settings* est présent, il doit avoir la même syntaxe que celle utilisée pour `theme_settings()`.

theme_settings(*themename*, *settings*)

Définit temporairement le thème actuel sur *themename*, applique les *settings* spécifiés, puis restaure le thème précédent.

Chaque clé dans *settings* est un style et chaque valeur peut contenir les clés `'configure'`, `'map'`, `'layout'` et `'element create'` et elles doivent avoir le même format que celui spécifié par les méthodes `Style.configure()`, `Style.map()`, `Style.layout()` et `Style.element_create()` respectivement.

À titre d'exemple, changeons un peu la *Combobox* pour le thème par défaut :

```
from tkinter import ttk
import tkinter

root = tkinter.Tk()

style = ttk.Style()
style.theme_settings("default", {
   "TCombobox": {
       "configure": {"padding": 5},
       "map": {
           "background": [("active", "green2"),
                          ("!disabled", "green4")],
           "fieldbackground": [("!disabled", "green3")],
           "foreground": [("focus", "OliveDrab1"),
                          ("!disabled", "OliveDrab2")]
       }
   }
})

combo = ttk.Combobox().pack()

root.mainloop()
```

theme_names()

Renvoie une liste de tous les thèmes connus.

theme_use(*themename=None*)

Si *themename* n'est pas donné, renvoie le thème utilisé. Sinon, définit le thème actuel sur *themename*, actualise tous les widgets et lève un événement «ThemeChanged».

Dispositions de mise en page

A layout can be just `None`, if it takes no options, or a dict of options specifying how to arrange the element. The layout mechanism uses a simplified version of the pack geometry manager : given an initial cavity, each element is allocated a parcel.

Les options/valeurs valides sont :

***side* : whichside**
Spécifie de quel côté de l'espace vide placer l'élément ; *top* (en haut), *right* (à droite), *bottom* (en bas) ou *left* (à gauche). S'il est omis, l'élément occupe toute l'espace.

***sticky* : nswe**
Spécifie où l'élément est placé à l'intérieur de sa partie d'espace alloué.

***unit* : 0 or 1**
Si défini sur 1, l'élément et tous ses descendants sont traités comme un seul élément aux fins de `Widget.identify()` et consœurs. Il est utilisé pour des choses comme la barre de défilement ou les agrandissements.

***children* : [sublayout...]**
Spécifie une liste d'éléments à placer à l'intérieur de l'élément. Chaque élément est un *n*-uplet (ou un autre type de séquence) où le premier élément est le nom de la mise en page et l'autre est un *Layout*.

26.9 IDLE --- Python editor and shell

Code source : Lib/idlelib/

IDLE est l'environnement de développement et d'apprentissage intégré de Python (*Integrated Development and Learning Environment*).

IDLE a les fonctionnalités suivantes :

- multiplateformes : fonctionne de la même manière sous Windows, *Unix* et *macOS*
- Console Python (interpréteur interactif) avec coloration du code entré, des sorties et des messages d'erreur
- éditeur de texte multi-fenêtres avec annulations multiples, coloration Python, indentation automatique, aide pour les appels de fonction, *autocomplétion*, parmi d'autres fonctionnalités
- recherche dans n'importe quelle fenêtre, remplacement dans une fenêtre d'édition et recherche dans des fichiers multiples (*grep*)
- débogueur avec points d'arrêt persistants, pas-à-pas et visualisation des espaces de nommage locaux et globaux
- configuration, navigateur et d'autres fenêtres de dialogue

26.9.1 Menus

IDLE a deux principaux types de fenêtre, la fenêtre de console et la fenêtre d'édition. Il est possible d'avoir de multiples fenêtres d'édition ouvertes simultanément. Sous Windows et Linux, chacune a son propre menu. Chaque menu documenté ci-dessous indique à quel type de fenêtre il est associé.

Les fenêtre d'affichage, comme celles qui sont utilisées pour *Edit => Find in Files*, sont un sous-type de fenêtre d'édition. Elles possèdent actuellement le même menu principal mais un titre par défaut et un menu contextuel différents.

Sous *macOS*, il y a un menu d'application. Il change dynamiquement en fonction de la fenêtre active. Il a un menu *IDLE* et certaines entrées décrites ci-dessous sont déplacées conformément aux directives d'Apple.

Menu *File* (Console et Éditeur)

New File
Crée une nouvelle fenêtre d'édition.

Open...
Ouvre un fichier existant avec une fenêtre de dialogue pour l'ouverture.

Open Module...
Ouvre un module existant (cherche dans *sys.path*).

Recent Files
Ouvre une liste des fichiers récents. Cliquez sur l'un d'eux pour l'ouvrir.

Module Browser
Montre les fonctions, classes et méthodes dans une arborescence pour le fichier en cours d'édition. Dans la console, ouvre d'abord un module.

Path Browser
Affiche les dossiers de *sys.path*, les modules, fonctions, classes et méthodes dans une arborescence.

Save
Enregistre la fenêtre active sous le fichier associé, s'il existe. Les fenêtres qui ont été modifiées depuis leur ouverture ou leur dernier enregistrement ont un * avant et après le titre de la fenêtre. S'il n'y a aucun fichier associé, exécute *Save As* à la place.

Save As...
Save the current window with a Save As dialog. The file saved becomes the new associated file for the window. (If your file namager is set to hide extensions, the current extension will be omitted in the file name box. If the new filename has no '.', '.py' and '.txt' will be added for Python and text files, except that on macOS Aqua,'.py' is added for all files.)

Save Copy As...
Save the current window to different file without changing the associated file. (See Save As note above about filename extensions.)

Print Window
Imprime la fenêtre active avec l'imprimante par défaut.

Close Window
Close the current window (if an unsaved editor, ask to save ; if an unsaved Shell, ask to quit execution). Calling `exit()` or `close()` in the Shell window also closes Shell. If this is the only window, also exit IDLE.

Exit IDLE
Close all windows and quit IDLE (ask to save unsaved edit windows).

Menu *Edit* (console et éditeur)

Undo
Annule le dernier changement dans la fenêtre active. Un maximum de 1000 changements peut être annulé.

Redo
Ré-applique le dernier changement annulé dans la fenêtre active.

Select All
Sélectionne la totalité du contenu de la fenêtre active.

Cut
Copie la sélection dans le presse-papier global ; puis supprime la sélection.

Copy
Copie la sélection dans le presse-papier global.

Paste
Insère le contenu du presse-papier global dans la fenêtre active.

Les fonctions du presse-papier sont aussi disponibles dans les menus contextuels.

Find...
Ouvre une fenêtre de recherche avec de nombreuses options

Find Again
Répète la dernière recherche, s'il y en a une.

Find Selection
Cherche la chaîne sélectionnée, s'il y en a une.

Find in Files...
Ouvre une fenêtre de recherche de fichiers. Présente les résultats dans une nouvelle fenêtre d'affichage.

Replace...
Ouvre une fenêtre de recherche et remplacement.

Go to Line
Move the cursor to the beginning of the line requested and make that line visible. A request past the end of the file goes to the end. Clear any selection and update the line and column status.

Show Completions
Open a scrollable list allowing selection of existing names. See *Completions* in the Editing and navigation section below.

Expand Word
Complète un préfixe que vous avez saisi pour correspondre à un mot complet de la même fenêtre ; recommencez pour obtenir un autre complément.

Show Call Tip
Après une parenthèse ouverte pour une fonction, ouvre une petite fenêtre avec des indications sur les paramètres de la fonction. Reportez-vous à *Aides aux appels* dans la section Édition et navigation ci-dessous.

Show Surrounding Parens
Surligne les parenthèses encadrantes.

Menu *Format* (fenêtre d'édition uniquement)

Format Paragraph
Reformate le paragraphe actif, délimité par des lignes vides, en un bloc de commentaires, ou la chaîne de caractères multi-lignes ou ligne sélectionnée en chaîne de caractères. Toutes les lignes du paragraphe seront formatées à moins de N colonnes, avec N valant 72 par défaut.

Indent Region
Décale les lignes sélectionnées vers la droite d'un niveau d'indentation (4 espaces par défaut).

Dedent Region
Décale les lignes sélectionnées vers la gauche d'un niveau d'indentation (4 espaces par défaut).

Comment Out Region
Insère ## devant les lignes sélectionnées.

Uncomment Region
Enlève les # ou ## au début des lignes sélectionnées.

Tabify Region
Transforme les blocs d'espaces *au début des lignes* en tabulations. (Note : Nous recommandons d'utiliser des blocs de 4 espaces pour indenter du code Python.)

Untabify Region
Transforme *toutes* les tabulations en le bon nombre d'espaces.

Toggle Tabs
Ouvre une boîte de dialogue permettant de passer des espaces aux tabulations (et inversement) pour l'indentation.

New Indent Width
Ouvre une boîte de dialogue pour changer la taille de l'indentation. La valeur par défaut acceptée par la communauté Python est de 4 espaces.

Strip Trailing Chitespace
Enlève les espaces après le dernier caractère non blanc d'une ligne en appliquant *str.rstrip* à chaque ligne, y compris les lignes avec des chaînes de caractères multi-lignes. À l'exception des fenêtres de terminal, supprime les lignes supplémentaires à la fin du fichier.

Menu *Run* (fenêtre d'édition uniquement)

Run Module
Applique *Check Module* (ci-dessus). S'il n'y a pas d'erreur, redémarre la console pour nettoyer l'environnement, puis exécute le module. Les sorties sont affichées dans la fenêtre de console. Notez qu'une sortie requiert l'utilisation de `print` ou `write`. Quand l'exécution est terminée, la console reste active et affiche une invite de commande. À ce moment, vous pouvez explorer interactivement le résultat de l'exécution. Ceci est similaire à l'exécution d'un fichier avec `python -i fichier` sur un terminal.

Run... Customized
Similaire à *Run Module*, mais lance le module avec des paramètres personnalisés. Les *Command Line Arguments* se rajoutent à `sys.argv` comme s'ils étaient passés par la ligne de commande. Le module peut être lancé dans le terminal sans avoir à le redémarrer.

Check Module
Vérifie la syntaxe du module actuellement ouvert dans la fenêtre d'édition. Si le module n'a pas été enregistré,

IDLE va soit demander à enregistrer à l'utilisateur, soit enregistrer automatiquement, selon l'option sélectionnée dans l'onglet *General* de la fenêtre de configuration d'*IDLE*. S'il y a une erreur de syntaxe, l'emplacement approximatif est indiqué dans la fenêtre d'édition.

Console Python
Ouvre ou active la fenêtre de console Python.

Menu Shell (fenêtre de console uniquement)

View Last Restart
Fait défiler la fenêtre de console jusqu'au dernier redémarrage de la console.

Restart Shell
Restart the shell to clean the environment and reset display and exception handling.

Previous History
Parcours les commandes précédentes dans l'historique qui correspondent à l'entrée actuelle.

Next History
Parcours les commandes suivantes dans l'historique qui correspondent à l'entrée actuelle.

Interrupt Execution
Arrête un programme en cours d'exécution.

Menu *Debug* (fenêtre de console uniquement)

Go to File/Line
Cherche, sur la ligne active et la ligne en-dessous, un nom de fichier et un numéro de ligne. Le cas échéant, ouvre le fichier s'il n'est pas encore ouvert et montre la ligne. Utilisez ceci pour visualiser les lignes de code source référencées dans un *traceback* d'exception et les lignes trouvées par *Find in Files*. Également disponible dans le menu contextuel des fenêtres de console et d'affichage.

***Debugger* ([dés]activer)**
Quand cette fonctionnalité est activée, le code saisi dans la console ou exécuté depuis un Éditeur s'exécutera avec le débogueur. Dans l'Éditeur, des points d'arrêt peuvent être placés avec le menu contextuel. Cette fonctionnalité est encore incomplète et plus ou moins expérimentale.

Stack Viewer
Montre l'état de la pile au moment de la dernière erreur dans une arborescence, avec accès aux variables locales et globales.

Auto-open Stack Viewer
Active ou désactive l'ouverture automatique de l'afficheur de pile après une erreur non gérée.

Menu *Options* (console et éditeur)

Configure IDLE
Ouvre une fenêtre de configuration et change les préférences pour les éléments suivants : police, indentation, raccourcis clavier, thème de coloration du texte, taille et nature de la fenêtre au lancement, sources d'aide additionnelles et extensions. Sous *macOS*, ouvrez la fenêtre de configuration en sélectionnant *Preferences* dans le menu de l'application. Pour plus de détails, référez-vous à *Modifier les préférences* dans Aide et paramètres.

La plupart des paramètres de configuration s'appliquent à toutes les fenêtres, ouvertes ou non. Les éléments d'option ci-dessous s'appliquent uniquement à la fenêtre active.

***Show/Hide Code Context* (fenêtres d'édition uniquement)**
Fais passer la fenêtre de taille normale à maximale. La taille de départ par défaut est de 40 lignes par 80 caractères, sauf changement dans l'onglet *General* de la fenêtre de configuration d'*IDLE*. Consultez *Code Context* dans la section « Édition et navigation » ci-dessous.

***Show/Hide Line Numbers* (fenêtre de l'éditeur uniquement)**
Ajoute une colonne à gauche de la fenêtre d'édition qui indique le numéro de chaque ligne de texte. Cette colonne est désactivée par défaut, ce qui peut être modifié dans les préférences (voir *Modifier les préférences*).

Zoom/Restore Height
Bascule la fenêtre entre la taille normale et la hauteur maximale. La taille initiale par défaut est de 40 lignes par 80 caractères, sauf si elle est modifiée dans l'onglet « *General* » de la boîte de dialogue « Configure IDLE ». La hauteur maximale d'un écran est déterminée en maximisant momentanément une fenêtre lors du premier zoom sur l'écran. La modification des paramètres d'écran peut invalider la hauteur enregistrée. Cette bascule n'a aucun effet lorsqu'une fenêtre est agrandie.

Menu *Windows* (console et éditeur)

Liste les noms de toutes les fenêtres ouvertes ; sélectionnez-en une pour l'amener au premier plan (en l'ouvrant si nécessaire).

Menu *Help* (console et éditeur)

About *IDLE*
Affiche la version, les copyrights, la licence, les crédits, entre autres.

IDLE Help
Affiche ce document *IDLE*, qui détaille les options des menus, les bases de l'édition et de la navigation ainsi que d'autres astuces.

Python Docs
Accède à la documentation Python locale, si installée, ou ouvre docs.python.org dans un navigateur pour afficher la documentation Python la plus récente.

Turtle Demo
Exécute le module *turtledemo* avec des exemples de code Python et de dessins *turtle*.

Des sources d'aide supplémentaires peuvent être ajoutées ici avec la fenêtre de configuration d'*IDLE* dans l'onglet *General*. Référez-vous à la sous-section *Sources d'aide* ci-dessous pour plus de détails sur les choix du menu d'aide.

Context menus

Vous pouvez ouvrir un menu contextuel par un clic droit dans une fenêtre (Contrôle-clic sous *macOS*). Les menus contextuels ont les fonctions de presse-papier standard, également disponibles dans le menu *Edit*.

Cut
Copie la sélection dans le presse-papier global ; puis supprime la sélection.

Copy
Copie la sélection dans le presse-papier global.

Paste
Insère le contenu du presse-papier global dans la fenêtre active.

Les fenêtres d'édition ont aussi des fonctions de points d'arrêt. Les lignes sur lesquelles est défini un point d'arrêt sont marquées de manière spéciale. Les points d'arrêt n'ont d'effet que lorsque l'exécution se déroule sous débogueur. Les points d'arrêt pour un fichier sont enregistrés dans le dossier `.idlerc` de l'utilisateur.

Set Breakpoint
Place un point d'arrêt sur la ligne active.

Clear Breakpoint
Enlève le point d'arrêt sur cette ligne.

Les fenêtres de console et d'affichage disposent en plus des éléments suivants.

Go to file/line
Même effet que dans le menu *Debug*.

Les fenêtres de console ont également une fonction de réduction des sorties détaillée dans la sous-section *fenêtre de console de Python* ci-dessous.

Squeeze
Si le curseur est sur une ligne d'affichage, compacte toute la sortie entre le code au-dessus et l'invite en-dessous en un bouton *"Squeezed text"*.

26.9.2 Editing and Navigation

Fenêtre d'édition

IDLE peut ouvrir une fenêtre d'édition quand il démarre, selon les paramètres et la manière dont vous démarrez *IDLE*. Ensuite, utilisez le menu *File*. Il ne peut y avoir qu'une fenêtre d'édition pour un fichier donné.

La barre de titre contient le nom du fichier, le chemin absolu et la version de Python et d'*IDLE* s'exécutant dans la fenêtre. La barre de statut contient le numéro de ligne ("*Ln*") et le numéro de la colonne ("*Col*"). Les numéros de ligne commencent à 1 ; les numéros de colonne commencent à 0.

IDLE suppose que les fichiers avec une extension en *.py** reconnue contiennent du code Python, mais pas les autres fichiers. Exécutez du code Python avec le menu *Run*.

Raccourcis clavier

The IDLE insertion cursor is a thin vertical bar between character positions. When characters are entered, the insertion cursor and everything to its right moves right one character and the new character is entered in the new space.

Several non-character keys move the cursor and possibly delete characters. Deletion does not puts text on the clipboard, but IDLE has an undo list. Wherever this doc discusses keys, 'C' refers to the `Control` key on Windows and Unix and the `Command` key on macOS. (And all such discussions assume that the keys have not been re-bound to something else.)

- Arrow keys move the cursor one character or line.
- `C-LeftArrow` and `C-RightArrow` moves left or right one word.
- `Home` and `End` go to the beginning or end of the line.
- `Page Up` and `Page Down` go up or down one screen.
- `C-Home` and `C-End` go to beginning or end of the file.
- `Backspace` and `Del` (or `C-d`) delete the previous or next character.
- `C-Backspace` and `C-Del` delete one word left or right.
- `C-k` deletes ('kills') everything to the right.

Les raccourcis clavier standards (comme `C-c` pour copier et `C-v` pour coller) peuvent fonctionner. Les raccourcis clavier sont sélectionnés dans la fenêtre de configuration d'*IDLE*.

Indentation automatique

Après une structure d'ouverture de bloc, la prochaine ligne est indentée de 4 espaces (dans la console Python d'une tabulation). Après certains mots-clefs (*break*, *return* etc) la ligne suivante est *dédentée*. Dans une indentation au début de la ligne, `Retour arrière` supprime jusqu'à 4 espaces s'il y en a. `Tab` insère des espaces (dans la console, une tabulation), en nombre dépendant de la configuration. Les tabulations sont actuellement restreintes à quatre espaces à cause de limitations de *Tcl/Tk*.

Cf. les commandes *indent/dedent region* dans le *menu *Format**.

Recherche et substitution

Any selection becomes a search target. However, only selections within a line work because searches are only performed within lines with the terminal newline removed. If `[x] Regular expression` is checked, the target is interpreted according to the Python re module.

Complétions

Completions are supplied, when requested and available, for module names, attributes of classes or functions, or filenames. Each request method displays a completion box with existing names. (See tab completions below for an exception.) For any box, change the name being completed and the item highlighted in the box by typing and deleting characters ; by hitting `Up`, `Down`, `PageUp`, `PageDown`, `Home`, and `End` keys ; and by a single click within the box. Close the box with `Escape`, `Enter`, and double `Tab` keys or clicks outside the box. A double click within the box selects and closes.

One way to open a box is to type a key character and wait for a predefined interval. This defaults to 2 seconds ; customize it in the settings dialog. (To prevent auto popups, set the delay to a large number of milliseconds, such as 100000000.) For imported module names or class or function attributes, type '.'. For filenames in the root directory,

type `os.sep` or `os.altsep` immediately after an opening quote. (On Windows, one can specify a drive first.) Move into subdirectories by typing a directory name and a separator.

Instead of waiting, or after a box is closed, open a completion box immediately with Show Completions on the Edit menu. The default hot key is `C-space`. If one types a prefix for the desired name before opening the box, the first match or near miss is made visible. The result is the same as if one enters a prefix after the box is displayed. Show Completions after a quote completes filenames in the current directory instead of a root directory.

Hitting `Tab` after a prefix usually has the same effect as Show Completions. (With no prefix, it indents.) However, if there is only one match to the prefix, that match is immediately added to the editor text without opening a box.

Invoking 'Show Completions', or hitting `Tab` after a prefix, outside of a string and without a preceding '.' opens a box with keywords, builtin names, and available module-level names.

When editing code in an editor (as oppose to Shell), increase the available module-level names by running your code and not restarting the Shell thereafter. This is especially useful after adding imports at the top of a file. This also increases possible attribute completions.

Completion boxes initially exclude names beginning with '_' or, for modules, not included in '__all__'. The hidden names can be accessed by typing '_' after '.', either before or after the box is opened.

Info-bulles

A calltip is shown automatically when one types `(` after the name of an *accessible* function. A function name expression may include dots and subscripts. A calltip remains until it is clicked, the cursor is moved out of the argument area, or `)` is typed. Whenever the cursor is in the argument part of a definition, select Edit and "Show Call Tip" on the menu or enter its shortcut to display a calltip.

The calltip consists of the function's signature and docstring up to the latter's first blank line or the fifth non-blank line. (Some builtin functions lack an accessible signature.) A '/' or '*' in the signature indicates that the preceding or following arguments are passed by position or name (keyword) only. Details are subject to change.

In Shell, the accessible functions depends on what modules have been imported into the user process, including those imported by Idle itself, and which definitions have been run, all since the last restart.

For example, restart the Shell and enter `itertools.count(`. A calltip appears because Idle imports itertools into the user process for its own use. (This could change.) Enter `turtle.write(` and nothing appears. Idle does not itself import turtle. The menu entry and shortcut also do nothing. Enter `import turtle`. Thereafter, `turtle.write(` will display a calltip.

In an editor, import statements have no effect until one runs the file. One might want to run a file after writing import statements, after adding function definitions, or after opening an existing file.

Contexte du code

Dans une fenêtre d'édition contenant du code Python, le contexte du code peut être activé pour afficher ou cacher une zone en haut de la fenêtre. Quand elle est affichée, cette zone gèle les lignes ouvrant le bloc de code, comme celles qui commencent par les mots-clés `class`, `def` ou `if`, qui auraient autrement été cachées plus haut dans le fichier. La taille de cette zone varie automatiquement selon ce qui est nécessaire pour afficher tous les niveaux de contexte, jusqu'à un nombre maximal de lignes défini dans la fenêtre de configuration d'*IDLE* (valeur qui vaut 15 par défaut). S'il n'y a pas de lignes de contexte et que cette fonctionnalité est activée, une unique ligne vide est affichée. Un clic sur une ligne dans la zone de contexte déplace cette ligne en haut de l'éditeur.

Les couleurs de texte et du fond pour la zone de contexte peuvent être configurées dans l'onglet *Highlights* de la fenêtre de configuration d'*IDLE*.

Shell window

In IDLE's Shell, enter, edit, and recall complete statements. (Most consoles and terminals only work with a single physical line at a time).

Submit a single-line statement for execution by hitting `Return` with the cursor anywhere on the line. If a line is extended with Backslash (`\`), the cursor must be on the last physical line. Submit a multi-line compound statement by entering a blank line after the statement.

When one pastes code into Shell, it is not compiled and possibly executed until one hits `Return`, as specified above. One may edit pasted code first. If one pastes more than one statement into Shell, the result will be a `SyntaxError` when multiple statements are compiled as if they were one.

Lines containing `RESTART` mean that the user execution process has been re-started. This occurs when the user execution process has crashed, when one requests a restart on the Shell menu, or when one runs code in an editor window.

The editing features described in previous subsections work when entering code interactively. IDLE's Shell window also responds to the following :

- `C-c` attempts to interrupt statement execution (but may fail).
- `C-d` closes Shell if typed at a `>>>` prompt.
- `Alt-p` and `Alt-n` (`C-p` and `C-n` on macOS) retrieve to the current prompt the previous or next previously entered statement that matches anything already typed.
- `Return` while the cursor is on any previous statement appends the latter to anything already typed at the prompt.

Coloration du texte

IDLE affiche par défaut le texte en noir sur blanc mais colore le texte qui possède une signification spéciale. Pour la console, ceci concerne les sorties de la console et de l'utilisateur ainsi que les erreurs de l'utilisateur. Pour le code Python, dans l'invite de commande de la console ou sur un éditeur, ce sont les mots-clefs, noms de fonctions et de classes incluses par défaut, les noms suivant `class` et `def`, les chaînes de caractères et les commentaires. Pour n'importe quelle fenêtre de texte, ce sont le curseur (si présent), le texte trouvé (s'il y en a) et le texte sélectionné.

IDLE also highlights the soft keywords `match`, `case`, and _ in pattern-matching statements. However, this highlighting is not perfect and will be incorrect in some rare cases, including some _-s in `case` patterns.

La coloration du texte est faite en arrière-plan, donc du texte non coloré est parfois visible. Pour changer les couleurs, utilisez l'onglet *Highlighting* de la fenêtre de configuration d'*IDLE*. Le marquage des points d'arrêt du débogueur dans l'éditeur et du texte dans les dialogues n'est pas configurable.

26.9.3 Startup and Code Execution

Quand il est démarré avec l'option `-s`, *IDLE* exécutera le fichier référencé par la variable d'environnement `IDLE*STARTUP` ou `PYTHONSTARTUP`. *IDLE* cherche d'abord `IDLESTARTUP` ; si `IDLESTARTUP` est présent, le fichier référencé est exécuté. Si `IDLESTARTUP` n'est pas présent, alors *IDLE* cherche `PYTHONSTARTUP`. Les fichiers référencés par ces variables d'environnement sont de bons endroits pour stocker des fonctions qui sont utilisées fréquemment depuis la console d'*IDLE* ou pour exécuter des commandes d'importation des modules communs.

De plus, `Tk` charge lui aussi un fichier de démarrage s'il est présent. Notez que le fichier de *Tk* est chargé sans condition. Ce fichier additionnel est `.Idle.py` et est recherché dans le dossier personnel de l'utilisateur. Les commandes dans ce fichier sont exécutées dans l'espace de nommage de *Tk*, donc ce fichier n'est pas utile pour importer des fonctions à utiliser depuis la console Python d'*IDLE*.

Utilisation de la ligne de commande

```
idle.py [-c command] [-d] [-e] [-h] [-i] [-r file] [-s] [-t title] [-] [arg] ...

-c command  run command in the shell window
-d          enable debugger and open shell window
-e          open editor window
-h          print help message with legal combinations and exit
-i          open shell window
-r file     run file in shell window
-s          run $IDLESTARTUP or $PYTHONSTARTUP first, in shell window
-t title    set title of shell window
-           run stdin in shell (- must be last option before args)
```

S'il y a des arguments :

— Si `-`, `-c` ou `-r` sont utilisés, tous les arguments sont placés dans `sys.argv[1:...]` et `sys.argv[0]` est assigné à `''`, `'-c'`, ou `'-r'`. Aucune fenêtre d'édition n'est ouverte, même si c'est le comportement par défaut fixé dans la fenêtre d'options.
— Sinon, les arguments sont des fichiers ouverts pour édition et `sys.argv` reflète les arguments passés à *IDLE* lui-même.

Échec au démarrage

IDLE uses a socket to communicate between the IDLE GUI process and the user code execution process. A connection must be established whenever the Shell starts or restarts. (The latter is indicated by a divider line that says 'RESTART'). If the user process fails to connect to the GUI process, it usually displays a `Tk` error box with a 'cannot connect' message that directs the user here. It then exits.

One specific connection failure on Unix systems results from misconfigured masquerading rules somewhere in a system's network setup. When IDLE is started from a terminal, one will see a message starting with `** Invalid host:`. The valid value is `127.0.0.1 (idlelib.rpc.LOCALHOST)`. One can diagnose with `tcpconnect -irv 127.0.0.1 6543` in one terminal window and `tcplisten <same args>` in another.

Une cause d'échec courant est un fichier écrit par l'utilisateur avec le même nom qu'un module de la bibliothèque standard, comme *random.py* et *tkinter.py*. Quand un fichier de ce genre est enregistré dans le même répertoire qu'un fichier à exécuter, *IDLE* ne peut pas importer le fichier standard. La solution actuelle consiste à renommer le fichier de l'utilisateur.

Même si c'est plus rare qu'avant, un antivirus ou un pare-feu peuvent interrompre la connexion. Si le programme ne peut pas être paramétré pour autoriser la connexion, alors il doit être éteint pour qu'*IDLE* puisse fonctionner. Cette connexion interne est sûre car aucune donnée n'est visible depuis un port extérieur. Un problème similaire est une mauvaise configuration du réseau qui bloque les connexions.

Des problèmes d'installation de Python stoppent parfois *IDLE* : il peut y avoir un conflit de versions ou bien l'installation peut nécessiter des privilèges administrateurs. Si on corrige le conflit , ou qu'on ne peut ou ne veut pas accorder de privilège, il peut être plus facile de désinstaller complètement Python et de recommencer.

A zombie pythonw.exe process could be a problem. On Windows, use Task Manager to check for one and stop it if there is. Sometimes a restart initiated by a program crash or Keyboard Interrupt (control-C) may fail to connect. Dismissing the error box or using Restart Shell on the Shell menu may fix a temporary problem.

When IDLE first starts, it attempts to read user configuration files in `~/.idlerc/` (~ is one's home directory). If there is a problem, an error message should be displayed. Leaving aside random disk glitches, this can be prevented by never editing the files by hand. Instead, use the configuration dialog, under Options. Once there is an error in a user configuration file, the best solution may be to delete it and start over with the settings dialog.

If IDLE quits with no message, and it was not started from a console, try starting it from a console or terminal (`python -m idlelib`) and see if this results in an error message.

On Unix-based systems with tcl/tk older than `8.6.11` (see `About IDLE`) certain characters of certain fonts can cause a tk failure with a message to the terminal. This can happen either if one starts IDLE to edit a file with such a character or later when entering such a character. If one cannot upgrade tcl/tk, then re-configure IDLE to use a font that works better.

Exécuter le code de l'utilisateur

With rare exceptions, the result of executing Python code with IDLE is intended to be the same as executing the same code by the default method, directly with Python in a text-mode system console or terminal window. However, the different interface and operation occasionally affect visible results. For instance, `sys.modules` starts with more entries, and `threading.active_count()` returns 2 instead of 1.

Par défaut, *IDLE* exécute le code de l'utilisateur dans un processus système séparé plutôt que dans le processus d'interface utilisateur qui exécute la console et l'éditeur. Dans le processus d'exécution, il remplace `sys.stdin`, `sys.stdout` et `sys.stderr` par des objets qui récupèrent les entrées et envoient les sorties à la fenêtre de console. Les valeurs originales stockées dans `sys.__stdin__`, `sys.__stdout__` et `sys.__stderr__` ne sont pas touchées, mais peuvent être `None`.

Sending print output from one process to a text widget in another is slower than printing to a system terminal in the same process. This has the most effect when printing multiple arguments, as the string for each argument, each

separator, the newline are sent separately. For development, this is usually not a problem, but if one wants to print faster in IDLE, format and join together everything one wants displayed together and then print a single string. Both format strings and `str.join()` can help combine fields and lines.

IDLE's standard stream replacements are not inherited by subprocesses created in the execution process, whether directly by user code or by modules such as multiprocessing. If such subprocess use `input` from sys.stdin or `print` or `write` to sys.stdout or sys.stderr, IDLE should be started in a command line window. (On Windows, use `python` or `py` rather than `pythonw` or `pyw`.) The secondary subprocess will then be attached to that window for input and output.

Si `sys` est réinitialisé par le code de l'utilisateur, comme avec `importlib.reload(sys)`, les changements d'*IDLE* seront perdus et l'entrée du clavier et la sortie à l'écran ne fonctionneront pas correctement.

Quand la console est au premier plan, elle contrôle le clavier et l'écran. Ceci est normalement transparent, mais les fonctions qui accèdent directement au clavier et à l'écran ne fonctionneront pas. Ceci inclut des fonctions spécifiques du système qui déterminent si une touche a été pressée et, le cas échéant, laquelle.

Le code IDLE tournant dans le processus d'exécution ajoute des appels de fonctions à la pile d'appels qui ne seraient pas là autrement. IDLE encapsule `sys.getrecursionlimit` et `sys.setrecursionlimit` pour réduire l'effet des piles de fonctions supplémentaires.

Lorsque l'utilisateur lève `SystemExit` directement ou en appelant `sys.exit`, IDLE revient au terminal IDLE au lieu de quitter.

Sortie de l'utilisateur sur la console

Quand un programme affiche du texte, le résultat est déterminé par le support d'affichage correspondant. Quand *IDLE* exécute du code de l'utilisateur, `sys.stdout` et `sys.stderr` sont connectées à la zone d'affichage de la console d'*IDLE*. Certaines de ces fonctionnalités sont héritées des widgets *Tk* sous-jacents. D'autres sont des additions programmées. Quand cela importe, la console est conçue pour le développement plutôt que l'exécution en production.

Par exemple, la console ne supprime jamais de sortie. Un programme qui écrit à l'infini dans la console finira par remplir la mémoire, ce qui entraînera un erreur mémoire. Par ailleurs, certains systèmes de fenêtres textuelles ne conservent que les n dernières lignes de sortie. Une console Windows, par exemple, conserve une quantité de lignes configurable entre 1 et 9999, avec une valeur par défaut de 300.

Un widget *Text* de *Tk* et donc la console d'*IDLE*, affiche des caractères (points de code) dans le sous-ensemble *BMP* (*Basic Multilingual Plane*) d'Unicode. Quels caractères sont affichés avec le bon glyphe et lesquels sont affichés avec un caractère de remplacement dépend du système d'exploitation et des polices installées. Les caractères de tabulation font que le texte suivant commencera après le prochain taquet de tabulation (ils sont placés tous les 8 *"caractères"*). Les caractères saut de ligne font apparaître le texte suivant sur une nouvelle ligne. Les autres caractères de contrôle sont ignorés ou affichés sous forme d'espace, de boîte, ou d'autre chose, selon le système d'exploitation et la police (déplacer le curseur de texte sur un affichage de ce genre peut provoquer un comportement surprenant du point de vue de l'espacement).

```
>>> s = 'a\tb\a<\x02><\r>\bc\nd'  # Enter 22 chars.
>>> len(s)
14
>>> s  # Display repr(s)
'a\tb\x07<\x02><\r>\x08c\nd'
>>> print(s, end='')  # Display s as is.
# Result varies by OS and font.  Try it.
```

La fonction `repr` est utilisée pour l'affichage interactif de la valeur des expressions. Elle renvoie une version modifiée de la chaîne en entrée dans laquelle les codes de contrôle, certains points de code *BMP* et tous les points de code non *BMP* sont remplacés par des caractères d'échappement. Comme montré ci-dessus, ceci permet d'identifier les caractères dans une chaîne, quelle que soit la façon dont elle est affichée.

Les sorties standard et d'erreur sont généralement séparées (sur des lignes séparées) de l'entrée de code et entre elles. Elles ont chacune une coloration différente.

Pour les *traceback* de *SyntaxError*, le "^" habituel marquant l'endroit où l'erreur a été détectée est remplacé par la coloration et le surlignage du texte avec une erreur. Quand du code exécuté depuis un fichier cause d'autres exceptions,

un clic droit sur la ligne du *traceback* permet d'accéder à la ligne correspondante dans un éditeur *IDLE*. Le fichier est ouvert si nécessaire.

La console a une fonctionnalité spéciale pour réduire les lignes de sorties à une étiquette *"Squeezed text"*. Ceci est fait automatiquement pour un sortie de plus de N lignes (N = 50 par défaut). N peut être changé dans la section *PyShell* de la page *General* de la fenêtre de configuration. Les sorties avec moins de lignes peuvent être réduites par un clic droit sur la sortie. Ceci peut être utile sur des lignes si longues qu'elles ralentissent la navigation.

Les sorties réduites sont étendues sur place en double-cliquant sur l'étiquette Elles peuvent aussi être envoyées au presse-papier ou sur un fenêtre séparée par un clic-droit sur l'étiquette.

Développer des applications *tkinter*

IDLE est intentionnellement différent de Python standard dans le but de faciliter le développement des programmes *tkinter*. Saisissez `import *tkinter* as tk; root = tk.Tk()` avec Python standard, rien n'apparaît. Saisissez la même chose dans *IDLE* et une fenêtre *tk* apparaît. En Python standard, il faut également saisir `root.update()` pour voir la fenêtre. *IDLE* fait un équivalent mais en arrière-plan, environ 20 fois par seconde, soit environ toutes les 50 millisecondes. Ensuite, saisissez `b = tk.Button(root, text='button'); b.pack()`. De la même manière, aucun changement n'est visible en Python standard jusqu'à la saisie de `root.update()`.

La plupart des programmes *tkinter* exécutent `root.mainloop()`, qui d'habitude ne renvoie pas jusqu'à ce que l'application *tk* soit détruite. Si le programme est exécuté avec `python -i` ou depuis un éditeur *IDLE*, une invite de commande `>>>` n'apparaît pas tant que `mainloop()` ne termine pas, c'est-à-dire quand il ne reste plus rien avec lequel interagir.

Avec un programme *tkinter* exécuté depuis un éditeur *IDLE*, vous pouvez immédiatement commenter l'appel à *mainloop*. On a alors accès à une invite de commande et on peut interagir en direct avec l'application. Il faut juste se rappeler de réactiver l'appel à *mainloop* lors de l'exécution en Python standard.

Exécution sans sous-processus

By default, IDLE executes user code in a separate subprocess via a socket, which uses the internal loopback interface. This connection is not externally visible and no data is sent to or received from the internet. If firewall software complains anyway, you can ignore it.

Si la tentative de connexion par le *socket* échoue, *IDLE* vous le notifie. Ce genre d'échec est parfois temporaire, mais s'il persiste, le problème peut soit venir d'un pare-feu qui bloque la connexion ou d'une mauvaise configuration dans un système particulier. Jusqu'à ce que le problème soit résolu, vous pouvez exécuter *IDLE* avec l'option *-n* de la ligne de commande.

Si *IDLE* est démarré avec l'option *-n* de la ligne de commande, il s'exécute dans un seul processus et ne crée pas de sous-processus pour exécuter le serveur RPC d'exécution de Python. Ceci peut être utile si Python ne peut pas créer de sous-processus ou de connecteur *RPC* sur votre plateforme. Cependant, dans ce mode, le code de l'utilisateur n'est pas isolé de *IDLE* lui-même. De plus, l'environnement n'est pas réinitialisé quand *Run/Run Module* (`F5`) est sélectionné. Si votre code a été modifié, vous devez *reload*() les modules affectés et ré-importer tous les éléments spécifiques (e.g. *from foo import baz*) pour que les changements prennent effet. Pour toutes ces raisons, il est préférable d'exécuter *IDLE* avec le sous-processus par défaut si c'est possible.

Obsolète depuis la version 3.4.

26.9.4 Help and Preferences

Sources d'aide

L'entrée du menu d'aide *"IDLE Help"* affiche une version *html* formatée du chapitre sur *IDLE* de la *Library Reference*. Le résultat, dans une fenêtre de texte *tkinter* en lecture-seule, est proche de ce qu'on voit dans un navigateur. Naviguez dans le texte avec la molette de la souris, la barre de défilement ou avec les touches directionnelles du clavier enfoncées. Ou cliquez sur le bouton TOC (*Table of Contents* : sommaire) et sélectionnez un titre de section dans l'espace ouvert.

L'entrée du menu d'aide *"Python Docs"* ouvre les sources d'aide détaillées, incluant des tutoriels, disponibles sur https://docs.python.org/x.y, avec *"x.y"* la version de Python en cours d'exécution. Si votre système a une copie locale de la documentation (cela peut être une option d'installation), c'est elle qui est ouverte.

Les URL sélectionnées peuvent être ajoutées ou enlevées du menu d'aide à n'importe quel moment en utilisant l'onglet « *General* » de la fenêtre « Configure IDLE ».

Modifier les préférences

Les préférences de polices, surlignage, touches et les préférences générales peuvent être changées via « Configure IDLE » dans le menu « Options ». Les paramètres modifiés par l'utilisateur sont enregistrés dans un dossier `.idlerc` dans le dossier personnel de l'utilisateur. Les problèmes causés par des fichiers de configuration utilisateur corrompus sont résolus en modifiant ou en supprimant un ou plusieurs fichiers dans *.idlerc*.

Dans l'onglet *Fonts*, regardez les échantillons de texte pour voir l'effet de la police et de la taille sur de multiples caractères de multiples langues. Éditez les échantillons pour ajouter d'autres caractères qui vous intéressent. Utilisez les échantillons pour sélectionner les polices à largeur constante. Si certains caractères posent des difficultés dans la console ou l'éditeur, ajoutez-les en haut des échantillons et essayez de changer d'abord la taille, puis la fonte.

Dans les onglets *Highlights* et *Keys*, sélectionnez un ensemble de couleurs et de raccourcis pré-inclus ou personnalisé. Pour utiliser un ensemble de couleurs et de raccourcis récent avec une version d'*IDLE* plus ancienne, enregistrez-le en tant que nouveau thème ou ensemble de raccourcis personnalisé ; il sera alors accessible aux *IDLE* plus anciens.

IDLE sous *macOS*

Dans *System Preferences : Dock*, on peut mettre *"Prefer tabs when opening documents"* à la valeur *"Always"*. Ce paramètre n'est pas compatible avec le cadriciel *tk/tkinter* utilisé par *IDLE* et il casse quelques fonctionnalités d'*IDLE*.

Extensions

IDLE inclut un outil d'extensions. Les préférences pour les extensions peuvent être changées avec l'onglet Extensions de la fenêtre de préférences. Lisez le début de *config-extensions.def* dans le dossier *idlelib* pour plus d'informations. La seule extension actuellement utilisée par défaut est *zzdummy*, un exemple également utilisé pour les tests.

26.9.5 idlelib --- implementation of IDLE application

Source code : Lib/idlelib

The Lib/idlelib package implements the IDLE application. See the rest of this page for how to use IDLE.

The files in idlelib are described in idlelib/README.txt. Access it either in idlelib or click Help => About IDLE on the IDLE menu. This file also maps IDLE menu items to the code that implements the item. Except for files listed under 'Startup', the idlelib code is 'private' in sense that feature changes can be backported (see **PEP 434**).

CHAPITRE 27

Outils de développement

The modules described in this chapter help you write software. For example, the *pydoc* module takes a module and generates documentation based on the module's contents. The *doctest* and *unittest* modules contains frameworks for writing unit tests that automatically exercise code and verify that the expected output is produced.

La liste des modules documentés dans ce chapitre est :

27.1 typing — Prise en charge des annotations de type

Ajouté dans la version 3.5.

Code source : Lib/typing.py

> **Note**
>
> The Python runtime does not enforce function and variable type annotations. They can be used by third party tools such as *type checkers*, IDEs, linters, etc.

This module provides runtime support for type hints.

Consider the function below :

```
def surface_area_of_cube(edge_length: float) -> str:
    return f"The surface area of the cube is {6 * edge_length ** 2}."
```

The function `surface_area_of_cube` takes an argument expected to be an instance of *float*, as indicated by the *type hint* `edge_length: float`. The function is expected to return an instance of *str*, as indicated by the `-> str` hint.

While type hints can be simple classes like *float* or *str*, they can also be more complex. The *typing* module provides a vocabulary of more advanced type hints.

New features are frequently added to the `typing` module. The typing_extensions package provides backports of these new features to older versions of Python.

Voir aussi

"Typing cheat sheet"
A quick overview of type hints (hosted at the mypy docs)

"Type System Reference" section of the mypy docs
The Python typing system is standardised via PEPs, so this reference should broadly apply to most Python type checkers. (Some parts may still be specific to mypy.)

"Static Typing with Python"
Type-checker-agnostic documentation written by the community detailing type system features, useful typing related tools and typing best practices.

27.1.1 Specification for the Python Type System

The canonical, up-to-date specification of the Python type system can be found at "Specification for the Python type system".

27.1.2 Alias de type

A type alias is defined using the `type` statement, which creates an instance of *`TypeAliasType`*. In this example, `Vector` and `list[float]` will be treated equivalently by static type checkers :

```
type Vector = list[float]

def scale(scalar: float, vector: Vector) -> Vector:
    return [scalar * num for num in vector]

# passes type checking; a list of floats qualifies as a Vector.
new_vector = scale(2.0, [1.0, -4.2, 5.4])
```

Les alias de type sont utiles pour simplifier les signatures complexes. Par exemple :

```
from collections.abc import Sequence

type ConnectionOptions = dict[str, str]
type Address = tuple[str, int]
type Server = tuple[Address, ConnectionOptions]

def broadcast_message(message: str, servers: Sequence[Server]) -> None:
    ...

# The static type checker will treat the previous type signature as
# being exactly equivalent to this one.
def broadcast_message(
    message: str,
    servers: Sequence[tuple[tuple[str, int], dict[str, str]]]
) -> None:
    ...
```

The `type` statement is new in Python 3.12. For backwards compatibility, type aliases can also be created through simple assignment :

```
Vector = list[float]
```

Or marked with *`TypeAlias`* to make it explicit that this is a type alias, not a normal variable assignment :

```
from typing import TypeAlias

Vector: TypeAlias = list[float]
```

27.1.3 *NewType*

Utilisez la classe *NewType* pour créer des types distincts :

```
from typing import NewType

UserId = NewType('UserId', int)
some_id = UserId(524313)
```

Le vérificateur de types statiques traite le nouveau type comme s'il s'agissait d'une sous-classe du type original. C'est utile pour aider à détecter les erreurs logiques :

```
def get_user_name(user_id: UserId) -> str:
    ...

# passes type checking
user_a = get_user_name(UserId(42351))

# fails type checking; an int is not a UserId
user_b = get_user_name(-1)
```

Vous pouvez toujours effectuer toutes les opérations applicables à un entier (type `int`) sur une variable de type `UserId`, mais le résultat sera toujours de type `int`. Ceci vous permet de passer un `UserId` partout où un `int` est attendu, mais vous empêche de créer accidentellement un `UserId` d'une manière invalide :

```
# 'output' is of type 'int', not 'UserId'
output = UserId(23413) + UserId(54341)
```

Notez que ces contrôles ne sont exécutés que par le vérificateur de types statique. À l'exécution, l'instruction `Derived = NewType('Derived', Base)` fait de `Derived` une fonction qui renvoie immédiatement le paramètre que vous lui passez. Cela signifie que l'expression `Derived(some_value)` ne crée pas une nouvelle classe et n'introduit pas de surcharge au-delà de celle d'un appel de fonction normal.

Plus précisément, l'expression `some_value is Derived(some_value)` est toujours vraie au moment de l'exécution.

La création d'un sous-type de `Derived` est invalide :

```
from typing import NewType

UserId = NewType('UserId', int)

# Fails at runtime and does not pass type checking
class AdminUserId(UserId): pass
```

Il est néanmoins possible de créer un *NewType* basé sur un `NewType` « dérivé » :

```
from typing import NewType

UserId = NewType('UserId', int)

ProUserId = NewType('ProUserId', UserId)
```

et la vérification de type pour `ProUserId` fonctionne comme prévu.

Voir la **PEP 484** pour plus de détails.

> **Note**
>
> Recall that the use of a type alias declares two types to be *equivalent* to one another. Doing `type Alias = Original` will make the static type checker treat `Alias` as being *exactly equivalent* to `Original` in all cases. This is useful when you want to simplify complex type signatures.
>
> En revanche, `NewType` déclare qu'un type est un *sous-type* d'un autre. Écrire `Derived = NewType('Derived', Original)` fait que le vérificateur de type statique traite `Derived` comme une *sous-classe* de `Original`, ce qui signifie qu'une valeur de type `Original` ne peut être utilisée dans les endroits où une valeur de type `Derived` est prévue. C'est utile lorsque vous voulez éviter les erreurs logiques avec un coût d'exécution minimal.

Ajouté dans la version 3.5.2.

Modifié dans la version 3.10 : `NewType` is now a class rather than a function. As a result, there is some additional runtime cost when calling `NewType` over a regular function.

Modifié dans la version 3.11 : The performance of calling `NewType` has been restored to its level in Python 3.9.

27.1.4 Annotating callable objects

Functions -- or other *callable* objects -- can be annotated using *`collections.abc.Callable`* or deprecated *`typing.Callable`*. `Callable[[int], str]` signifies a function that takes a single parameter of type *`int`* and returns a *`str`*.

For example :

```
from collections.abc import Callable, Awaitable

def feeder(get_next_item: Callable[[], str]) -> None:
    ...  # Body

def async_query(on_success: Callable[[int], None],
                on_error: Callable[[int, Exception], None]) -> None:
    ...  # Body

async def on_update(value: str) -> None:
    ...  # Body

callback: Callable[[str], Awaitable[None]] = on_update
```

The subscription syntax must always be used with exactly two values : the argument list and the return type. The argument list must be a list of types, a *`ParamSpec`*, *`Concatenate`*, or an ellipsis. The return type must be a single type.

If a literal ellipsis `...` is given as the argument list, it indicates that a callable with any arbitrary parameter list would be acceptable :

```
def concat(x: str, y: str) -> str:
    return x + y

x: Callable[..., str]
x = str     # OK
x = concat  # Also OK
```

`Callable` cannot express complex signatures such as functions that take a variadic number of arguments, *overloaded functions*, or functions that have keyword-only parameters. However, these signatures can be expressed by defining a *`Protocol`* class with a `__call__()` method :

```
from collections.abc import Iterable
from typing import Protocol

class Combiner(Protocol):
    def __call__(self, *vals: bytes, maxlen: int | None = None) -> list[bytes]: ...

def batch_proc(data: Iterable[bytes], cb_results: Combiner) -> bytes:
    for item in data:
        ...

def good_cb(*vals: bytes, maxlen: int | None = None) -> list[bytes]:
    ...
def bad_cb(*vals: bytes, maxitems: int | None) -> list[bytes]:
    ...

batch_proc([], good_cb)  # OK
batch_proc([], bad_cb)   # Error! Argument 2 has incompatible type because of
                         # different name and kind in the callback
```

Les appelables qui prennent en argument d'autres appelables peuvent indiquer que leurs types de paramètres dépendent les uns des autres en utilisant *ParamSpec*. De plus, si un appelable ajoute ou supprime des arguments d'autres appelables, l'opérateur *Concatenate* peut être utilisé. Ils prennent la forme `Callable[ParamSpecVariable, ReturnType]` et `Callable[Concatenate[Arg1Type, Arg2Type, ..., ParamSpecVariable], ReturnType]` respectivement.

Modifié dans la version 3.10 : `Callable` prend désormais en charge *ParamSpec* et *Concatenate*. Voir **PEP 612** pour plus de détails.

Voir aussi

La documentation pour *ParamSpec* et *Concatenate* fournit des exemples d'utilisation dans `Callable`.

27.1.5 Génériques

Since type information about objects kept in containers cannot be statically inferred in a generic way, many container classes in the standard library support subscription to denote the expected types of container elements.

```
from collections.abc import Mapping, Sequence

class Employee: ...

# Sequence[Employee] indicates that all elements in the sequence
# must be instances of "Employee".
# Mapping[str, str] indicates that all keys and all values in the mapping
# must be strings.
def notify_by_email(employees: Sequence[Employee],
                    overrides: Mapping[str, str]) -> None: ...
```

Generic functions and classes can be parameterized by using type parameter syntax :

```
from collections.abc import Sequence

def first[T](l: Sequence[T]) -> T:  # Function is generic over the TypeVar "T"
    return l[0]
```

Or by using the *TypeVar* factory directly :

```
from collections.abc import Sequence
from typing import TypeVar

U = TypeVar('U')                  # Declare type variable "U"

def second(l: Sequence[U]) -> U:  # Function is generic over the TypeVar "U"
    return l[1]
```

Modifié dans la version 3.12 : Syntactic support for generics is new in Python 3.12.

27.1.6 Annotating tuples

For most containers in Python, the typing system assumes that all elements in the container will be of the same type. For example :

```
from collections.abc import Mapping

# Type checker will infer that all elements in ``x`` are meant to be ints
x: list[int] = []

# Type checker error: ``list`` only accepts a single type argument:
y: list[int, str] = [1, 'foo']

# Type checker will infer that all keys in ``z`` are meant to be strings,
# and that all values in ``z`` are meant to be either strings or ints
z: Mapping[str, str | int] = {}
```

`list` only accepts one type argument, so a type checker would emit an error on the `y` assignment above. Similarly, `Mapping` only accepts two type arguments : the first indicates the type of the keys, and the second indicates the type of the values.

Unlike most other Python containers, however, it is common in idiomatic Python code for tuples to have elements which are not all of the same type. For this reason, tuples are special-cased in Python's typing system. `tuple` accepts *any number* of type arguments :

```
# OK: ``x`` is assigned to a tuple of length 1 where the sole element is an int
x: tuple[int] = (5,)

# OK: ``y`` is assigned to a tuple of length 2;
# element 1 is an int, element 2 is a str
y: tuple[int, str] = (5, "foo")

# Error: the type annotation indicates a tuple of length 1,
# but ``z`` has been assigned to a tuple of length 3
z: tuple[int] = (1, 2, 3)
```

To denote a tuple which could be of *any* length, and in which all elements are of the same type `T`, use `tuple[T, ...]`. To denote an empty tuple, use `tuple[()]`. Using plain `tuple` as an annotation is equivalent to using `tuple[Any, ...]` :

```
x: tuple[int, ...] = (1, 2)
# These reassignments are OK: ``tuple[int, ...]`` indicates x can be of any length
x = (1, 2, 3)
x = ()
# This reassignment is an error: all elements in ``x`` must be ints
x = ("foo", "bar")

# ``y`` can only ever be assigned to an empty tuple
```

(suite sur la page suivante)

(suite de la page précédente)

```
y: tuple[()] = ()

z: tuple = ("foo", "bar")
# These reassignments are OK: plain ``tuple`` is equivalent to ``tuple[Any, ...]``
z = (1, 2, 3)
z = ()
```

27.1.7 The type of class objects

A variable annotated with `C` may accept a value of type `C`. In contrast, a variable annotated with `type[C]` (or deprecated *`typing.Type[C]`*) may accept values that are classes themselves -- specifically, it will accept the *class object* of `C`. For example :

```
a = 3         # Has type ``int``
b = int       # Has type ``type[int]``
c = type(a)   # Also has type ``type[int]``
```

Note that `type[C]` is covariant :

```
class User: ...
class ProUser(User): ...
class TeamUser(User): ...

def make_new_user(user_class: type[User]) -> User:
    # ...
    return user_class()

make_new_user(User)       # OK
make_new_user(ProUser)    # Also OK: ``type[ProUser]`` is a subtype of␣
↪``type[User]``
make_new_user(TeamUser)   # Still fine
make_new_user(User())     # Error: expected ``type[User]`` but got ``User``
make_new_user(int)        # Error: ``type[int]`` is not a subtype of ``type[User]``
```

The only legal parameters for *`type`* are classes, *`Any`*, *type variables*, and unions of any of these types. For example :

```
def new_non_team_user(user_class: type[BasicUser | ProUser]): ...

new_non_team_user(BasicUser)   # OK
new_non_team_user(ProUser)     # OK
new_non_team_user(TeamUser)    # Error: ``type[TeamUser]`` is not a subtype
                               # of ``type[BasicUser | ProUser]``
new_non_team_user(User)        # Also an error
```

`type[Any]` is equivalent to *`type`*, which is the root of Python's metaclass hierarchy.

27.1.8 Annotating generators and coroutines

A generator can be annotated using the generic type *`Generator[YieldType, SendType, ReturnType]`*. For example :

```
def echo_round() -> Generator[int, float, str]:
    sent = yield 0
    while sent >= 0:
        sent = yield round(sent)
    return 'Done'
```

Note that unlike many other generic classes in the standard library, the `SendType` of *Generator* behaves contravariantly, not covariantly or invariantly.

The `SendType` and `ReturnType` parameters default to `None` :

```
def infinite_stream(start: int) -> Generator[int]:
    while True:
        yield start
        start += 1
```

It is also possible to set these types explicitly :

```
def infinite_stream(start: int) -> Generator[int, None, None]:
    while True:
        yield start
        start += 1
```

Simple generators that only ever yield values can also be annotated as having a return type of either *Iterable[YieldType]* or *Iterator[YieldType]* :

```
def infinite_stream(start: int) -> Iterator[int]:
    while True:
        yield start
        start += 1
```

Async generators are handled in a similar fashion, but don't expect a `ReturnType` type argument (*AsyncGenerator[YieldType, SendType]*). The `SendType` argument defaults to `None`, so the following definitions are equivalent :

```
async def infinite_stream(start: int) -> AsyncGenerator[int]:
    while True:
        yield start
        start = await increment(start)

async def infinite_stream(start: int) -> AsyncGenerator[int, None]:
    while True:
        yield start
        start = await increment(start)
```

As in the synchronous case, *AsyncIterable[YieldType]* and *AsyncIterator[YieldType]* are available as well :

```
async def infinite_stream(start: int) -> AsyncIterator[int]:
    while True:
        yield start
        start = await increment(start)
```

Coroutines can be annotated using *Coroutine[YieldType, SendType, ReturnType]*. Generic arguments correspond to those of *Generator*, for example :

```
from collections.abc import Coroutine
c: Coroutine[list[str], str, int]  # Some coroutine defined elsewhere
x = c.send('hi')                   # Inferred type of 'x' is list[str]
async def bar() -> None:
    y = await c                    # Inferred type of 'y' is int
```

27.1.9 Types génériques définis par l'utilisateur

Une classe définie par l'utilisateur peut être définie comme une classe générique.

```
from logging import Logger

class LoggedVar[T]:
    def __init__(self, value: T, name: str, logger: Logger) -> None:
        self.name = name
        self.logger = logger
        self.value = value

    def set(self, new: T) -> None:
        self.log('Set ' + repr(self.value))
        self.value = new

    def get(self) -> T:
        self.log('Get ' + repr(self.value))
        return self.value

    def log(self, message: str) -> None:
        self.logger.info('%s: %s', self.name, message)
```

This syntax indicates that the class `LoggedVar` is parameterised around a single *type variable* `T` . This also makes `T` valid as a type within the class body.

Generic classes implicitly inherit from `Generic`. For compatibility with Python 3.11 and lower, it is also possible to inherit explicitly from `Generic` to indicate a generic class :

```
from typing import TypeVar, Generic

T = TypeVar('T')

class LoggedVar(Generic[T]):
    ...
```

Generic classes have `__class_getitem__()` methods, meaning they can be parameterised at runtime (e.g. `LoggedVar[int]` below) :

```
from collections.abc import Iterable

def zero_all_vars(vars: Iterable[LoggedVar[int]]) -> None:
    for var in vars:
        var.set(0)
```

A generic type can have any number of type variables. All varieties of `TypeVar` are permissible as parameters for a generic type :

```
from typing import TypeVar, Generic, Sequence

class WeirdTrio[T, B: Sequence[bytes], S: (int, str)]:
    ...

OldT = TypeVar('OldT', contravariant=True)
OldB = TypeVar('OldB', bound=Sequence[bytes], covariant=True)
OldS = TypeVar('OldS', int, str)

class OldWeirdTrio(Generic[OldT, OldB, OldS]):
    ...
```

Chaque argument de variable de type *Generic* doit être distinct. Ceci n'est donc pas valable :

```
from typing import TypeVar, Generic
...

class Pair[M, M]:  # SyntaxError
    ...

T = TypeVar('T')

class Pair(Generic[T, T]):   # INVALID
    ...
```

Generic classes can also inherit from other classes :

```
from collections.abc import Sized

class LinkedList[T](Sized):
    ...
```

When inheriting from generic classes, some type parameters could be fixed :

```
from collections.abc import Mapping

class MyDict[T](Mapping[str, T]):
    ...
```

Dans ce cas, `MyDict` a un seul paramètre, `T`.

Using a generic class without specifying type parameters assumes *Any* for each position. In the following example, `MyIterable` is not generic but implicitly inherits from `Iterable[Any]` :

```
from collections.abc import Iterable

class MyIterable(Iterable): # Same as Iterable[Any]
    ...
```

User-defined generic type aliases are also supported. Examples :

```
from collections.abc import Iterable

type Response[S] = Iterable[S] | int

# Return type here is same as Iterable[str] | int
def response(query: str) -> Response[str]:
    ...

type Vec[T] = Iterable[tuple[T, T]]

def inproduct[T: (int, float, complex)](v: Vec[T]) -> T: # Same as␣
↪Iterable[tuple[T, T]]
    return sum(x*y for x, y in v)
```

For backward compatibility, generic type aliases can also be created through a simple assignment :

```
from collections.abc import Iterable
from typing import TypeVar

S = TypeVar("S")
Response = Iterable[S] | int
```

Modifié dans la version 3.7 : *Generic* n'a plus de métaclasse personnalisée.

Modifié dans la version 3.12 : Syntactic support for generics and type aliases is new in version 3.12. Previously, generic classes had to explicitly inherit from *Generic* or contain a type variable in one of their bases.

User-defined generics for parameter expressions are also supported via parameter specification variables in the form `[**P]`. The behavior is consistent with type variables' described above as parameter specification variables are treated by the typing module as a specialized type variable. The one exception to this is that a list of types can be used to substitute a *ParamSpec* :

```
>>> class Z[T, **P]: ...  # T is a TypeVar; P is a ParamSpec
...
>>> Z[int, [dict, float]]
__main__.Z[int, [dict, float]]
```

Classes generic over a *ParamSpec* can also be created using explicit inheritance from *Generic*. In this case, `**` is not used :

```
from typing import ParamSpec, Generic

P = ParamSpec('P')

class Z(Generic[P]):
    ...
```

Another difference between *TypeVar* and *ParamSpec* is that a generic with only one parameter specification variable will accept parameter lists in the forms `X[[Type1, Type2, ...]]` and also `X[Type1, Type2, ...]` for aesthetic reasons. Internally, the latter is converted to the former, so the following are equivalent :

```
>>> class X[**P]: ...
...
>>> X[int, str]
__main__.X[[int, str]]
>>> X[[int, str]]
__main__.X[[int, str]]
```

Note that generics with *ParamSpec* may not have correct `__parameters__` after substitution in some cases because they are intended primarily for static type checking.

Modifié dans la version 3.10 : *Generic* can now be parameterized over parameter expressions. See *ParamSpec* and **PEP 612** for more details.

A user-defined generic class can have ABCs as base classes without a metaclass conflict. Generic metaclasses are not supported. The outcome of parameterizing generics is cached, and most types in the typing module are *hashable* and comparable for equality.

27.1.10 Le type `Any`

Un type particulier est *Any*. Un vérificateur de types statiques traite chaque type comme étant compatible avec *Any* et *Any* comme étant compatible avec chaque type.

This means that it is possible to perform any operation or method call on a value of type *Any* and assign it to any variable :

```
from typing import Any

a: Any = None
a = []          # OK
a = 2           # OK

s: str = ''
```

(suite sur la page suivante)

(suite de la page précédente)

```
s = a            # OK

def foo(item: Any) -> int:
    # Passes type checking; 'item' could be any type,
    # and that type might have a 'bar' method
    item.bar()
    ...
```

Notice that no type checking is performed when assigning a value of type `Any` to a more precise type. For example, the static type checker did not report an error when assigning `a` to `s` even though `s` was declared to be of type `str` and receives an `int` value at runtime !

De plus, toutes les fonctions sans type de retour ni type de paramètre sont considérées comme utilisant `Any` implicitement par défaut :

```
def legacy_parser(text):
    ...
    return data

# A static type checker will treat the above
# as having the same signature as:
def legacy_parser(text: Any) -> Any:
    ...
    return data
```

Ce comportement permet à `Any` d'être utilisé comme succédané lorsque vous avez besoin de mélanger du code typé dynamiquement et statiquement.

Comparons le comportement de `Any` avec celui de `object`. De la même manière que pour `Any`, chaque type est un sous-type de `object`. Cependant, contrairement à `Any`, l'inverse n'est pas vrai : `object` n'est *pas* un sous-type de chaque autre type.

Cela signifie que lorsque le type d'une valeur est `object`, un vérificateur de types rejette presque toutes les opérations sur celle-ci, et l'affecter à une variable (ou l'utiliser comme une valeur de retour) d'un type plus spécialisé est une erreur de typage. Par exemple :

```
def hash_a(item: object) -> int:
    # Fails type checking; an object does not have a 'magic' method.
    item.magic()
    ...

def hash_b(item: Any) -> int:
    # Passes type checking
    item.magic()
    ...

# Passes type checking, since ints and strs are subclasses of object
hash_a(42)
hash_a("foo")

# Passes type checking, since Any is compatible with all types
hash_b(42)
hash_b("foo")
```

Utilisez `object` pour indiquer qu'une valeur peut être de n'importe quel type de manière sûre. Utiliser `Any` pour indiquer qu'une valeur est typée dynamiquement.

27.1.11 Sous-typage nominal et sous-typage structurel

Initially **PEP 484** defined the Python static type system as using *nominal subtyping*. This means that a class `A` is allowed where a class `B` is expected if and only if `A` is a subclass of `B`.

This requirement previously also applied to abstract base classes, such as *`Iterable`*. The problem with this approach is that a class had to be explicitly marked to support them, which is unpythonic and unlike what one would normally do in idiomatic dynamically typed Python code. For example, this conforms to **PEP 484** :

```
from collections.abc import Sized, Iterable, Iterator

class Bucket(Sized, Iterable[int]):
    ...
    def __len__(self) -> int: ...
    def __iter__(self) -> Iterator[int]: ...
```

La **PEP 544** permet de résoudre ce problème en permettant aux utilisateurs d'écrire le code ci-dessus sans classes mères explicites dans la définition de classe, permettant à `Bucket` d'être implicitement considéré comme un sous-type de `Sized` et `Iterable[int]` par des vérificateurs de type statique. C'est ce qu'on appelle le *sous-typage structurel* (ou typage canard) :

```
from collections.abc import Iterator, Iterable

class Bucket:  # Note: no base classes
    ...
    def __len__(self) -> int: ...
    def __iter__(self) -> Iterator[int]: ...

def collect(items: Iterable[int]) -> int: ...
result = collect(Bucket())  # Passes type check
```

De plus, en sous-classant une classe spéciale *`Protocol`*, un utilisateur peut définir de nouveaux protocoles personnalisés pour profiter pleinement du sous-typage structurel (voir exemples ci-dessous).

27.1.12 Classe de données

The `typing` module defines the following classes, functions and decorators.

Special typing primitives

Special types

These can be used as types in annotations. They do not support subscription using `[]`.

`typing.`**`Any`**

Type spécial indiquant un type non contraint.

— Chaque type est compatible avec *`Any`*.

— *`Any`* est compatible avec tous les types.

Modifié dans la version 3.11 : *`Any`* can now be used as a base class. This can be useful for avoiding type checker errors with classes that can duck type anywhere or are highly dynamic.

`typing.`**`AnyStr`**

A *constrained type variable*.

Definition :

```
AnyStr = TypeVar('AnyStr', str, bytes)
```

`AnyStr` is meant to be used for functions that may accept *`str`* or *`bytes`* arguments but cannot allow the two to mix.

Par exemple :

```
def concat(a: AnyStr, b: AnyStr) -> AnyStr:
    return a + b

concat("foo", "bar")     # OK, output has type 'str'
concat(b"foo", b"bar")   # OK, output has type 'bytes'
concat("foo", b"bar")    # Error, cannot mix str and bytes
```

Note that, despite its name, `AnyStr` has nothing to do with the `Any` type, nor does it mean "any string". In particular, `AnyStr` and `str | bytes` are different from each other and have different use cases :

```
# Invalid use of AnyStr:
# The type variable is used only once in the function signature,
# so cannot be "solved" by the type checker
def greet_bad(cond: bool) -> AnyStr:
    return "hi there!" if cond else b"greetings!"

# The better way of annotating this function:
def greet_proper(cond: bool) -> str | bytes:
    return "hi there!" if cond else b"greetings!"
```

Deprecated since version 3.13, will be removed in version 3.18 : Deprecated in favor of the new type parameter syntax. Use `class A[T: (str, bytes)]: ...` instead of importing `AnyStr`. See **PEP 695** for more details.

In Python 3.16, `AnyStr` will be removed from `typing.__all__`, and deprecation warnings will be emitted at runtime when it is accessed or imported from `typing`. `AnyStr` will be removed from `typing` in Python 3.18.

`typing.`**`LiteralString`**

Special type that includes only literal strings.

Any string literal is compatible with `LiteralString`, as is another `LiteralString`. However, an object typed as just `str` is not. A string created by composing `LiteralString`-typed objects is also acceptable as a `LiteralString`.

Example :

```
def run_query(sql: LiteralString) -> None:
    ...

def caller(arbitrary_string: str, literal_string: LiteralString) -> None:
    run_query("SELECT * FROM students")  # OK
    run_query(literal_string)  # OK
    run_query("SELECT * FROM " + literal_string)  # OK
    run_query(arbitrary_string)  # type checker error
    run_query(  # type checker error
        f"SELECT * FROM students WHERE name = {arbitrary_string}"
    )
```

`LiteralString` is useful for sensitive APIs where arbitrary user-generated strings could generate problems. For example, the two cases above that generate type checker errors could be vulnerable to an SQL injection attack.

See **PEP 675** for more details.

Ajouté dans la version 3.11.

`typing.`**`Never`**

`typing.`**`NoReturn`**

`Never` and `NoReturn` represent the bottom type, a type that has no members.

They can be used to indicate that a function never returns, such as `sys.exit()` :

```
from typing import Never  # or NoReturn

def stop() -> Never:
    raise RuntimeError('no way')
```

Or to define a function that should never be called, as there are no valid arguments, such as *assert_never()* :

```
from typing import Never  # or NoReturn

def never_call_me(arg: Never) -> None:
    pass

def int_or_str(arg: int | str) -> None:
    never_call_me(arg)  # type checker error
    match arg:
        case int():
            print("It's an int")
        case str():
            print("It's a str")
        case _:
            never_call_me(arg)  # OK, arg is of type Never (or NoReturn)
```

`Never` and `NoReturn` have the same meaning in the type system and static type checkers treat both equivalently.

Ajouté dans la version 3.6.2 : Added *NoReturn*.

Ajouté dans la version 3.11 : Added *Never*.

`typing.`**`Self`**

Special type to represent the current enclosed class.

Par exemple :

```
from typing import Self, reveal_type

class Foo:
    def return_self(self) -> Self:
        ...
        return self

class SubclassOfFoo(Foo): pass

reveal_type(Foo().return_self())  # Revealed type is "Foo"
reveal_type(SubclassOfFoo().return_self())  # Revealed type is "SubclassOfFoo"
```

This annotation is semantically equivalent to the following, albeit in a more succinct fashion :

```
from typing import TypeVar

Self = TypeVar("Self", bound="Foo")

class Foo:
    def return_self(self: Self) -> Self:
        ...
        return self
```

In general, if something returns `self`, as in the above examples, you should use `Self` as the return annotation. If `Foo.return_self` was annotated as returning `"Foo"`, then the type checker would infer the object returned from `SubclassOfFoo.return_self` as being of type `Foo` rather than `SubclassOfFoo`.

Other common use cases include :
— *classmethod*s that are used as alternative constructors and return instances of the `cls` parameter.
— Annotating an `__enter__()` method which returns self.
You should not use `Self` as the return annotation if the method is not guaranteed to return an instance of a subclass when the class is subclassed :

```
class Eggs:
    # Self would be an incorrect return annotation here,
    # as the object returned is always an instance of Eggs,
    # even in subclasses
    def returns_eggs(self) -> "Eggs":
        return Eggs()
```

See **PEP 673** for more details.

Ajouté dans la version 3.11.

`typing.`**`TypeAlias`**

Special annotation for explicitly declaring a *type alias*.

Par exemple :

```
from typing import TypeAlias

Factors: TypeAlias = list[int]
```

`TypeAlias` is particularly useful on older Python versions for annotating aliases that make use of forward references, as it can be hard for type checkers to distinguish these from normal variable assignments :

```
from typing import Generic, TypeAlias, TypeVar

T = TypeVar("T")

# "Box" does not exist yet,
# so we have to use quotes for the forward reference on Python <3.12.
# Using ``TypeAlias`` tells the type checker that this is a type alias␣
↪declaration,
# not a variable assignment to a string.
BoxOfStrings: TypeAlias = "Box[str]"

class Box(Generic[T]):
    @classmethod
    def make_box_of_strings(cls) -> BoxOfStrings: ...
```

See **PEP 613** for more details.

Ajouté dans la version 3.10.

Obsolète depuis la version 3.12 : *`TypeAlias`* is deprecated in favor of the `type` statement, which creates instances of *`TypeAliasType`* and which natively supports forward references. Note that while *`TypeAlias`* and *`TypeAliasType`* serve similar purposes and have similar names, they are distinct and the latter is not the type of the former. Removal of *`TypeAlias`* is not currently planned, but users are encouraged to migrate to `type` statements.

Special forms

These can be used as types in annotations. They all support subscription using `[]`, but each has a unique syntax.

`typing.`**`Union`**

Union type ; `Union[X, Y]` is equivalent to `X | Y` and means either X or Y.

To define a union, use e.g. `Union[int, str]` or the shorthand `int | str`. Using that shorthand is recommended. Details :

— Les arguments doivent être des types et il doit y en avoir au moins un.

— Les unions d'unions sont aplanies, par exemple :

```
Union[Union[int, str], float] == Union[int, str, float]
```

— Les unions d'un seul argument disparaissent, par exemple :

```
Union[int] == int  # The constructor actually returns int
```

— Les arguments redondants sont ignorés, par exemple :

```
Union[int, str, int] == Union[int, str] == int | str
```

— Lors de la comparaison d'unions, l'ordre des arguments est ignoré, par exemple :

```
Union[int, str] == Union[str, int]
```

— You cannot subclass or instantiate a `Union`.

— Vous ne pouvez pas écrire `Union[X][Y]`.

Modifié dans la version 3.7 : Ne supprime pas les sous-classes explicites des unions à l'exécution.

Modifié dans la version 3.10 : Unions can now be written as `X | Y`. See *union type expressions*.

`typing.`**`Optional`**

`Optional[X]` is equivalent to `X | None` (or `Union[X, None]`).

Notez que ce n'est pas le même concept qu'un argument optionnel, qui est un argument qui possède une valeur par défaut. Un argument optionnel (qui a une valeur par défaut) ne nécessite pas, à ce titre, le qualificatif `Optional` sur son annotation de type. Par exemple :

```
def foo(arg: int = 0) -> None:
    ...
```

Par contre, si une valeur explicite de `None` est permise, l'utilisation de `Optional` est appropriée, que l'argument soit facultatif ou non. Par exemple :

```
def foo(arg: Optional[int] = None) -> None:
    ...
```

Modifié dans la version 3.10 : Optional can now be written as `X | None`. See *union type expressions*.

`typing.`**`Concatenate`**

Special form for annotating higher-order functions.

`Concatenate` can be used in conjunction with *Callable* and `ParamSpec` to annotate a higher-order callable which adds, removes, or transforms parameters of another callable. Usage is in the form `Concatenate[Arg1Type, Arg2Type, ..., ParamSpecVariable]`. `Concatenate` is currently only valid when used as the first argument to a *Callable*. The last parameter to `Concatenate` must be a `ParamSpec` or ellipsis (`...`).

For example, to annotate a decorator `with_lock` which provides a `threading.Lock` to the decorated function, `Concatenate` can be used to indicate that `with_lock` expects a callable which takes in a `Lock` as the first argument, and returns a callable with a different type signature. In this case, the `ParamSpec` indicates that the returned callable's parameter types are dependent on the parameter types of the callable being passed in :

```
from collections.abc import Callable
from threading import Lock
from typing import Concatenate

# Use this lock to ensure that only one thread is executing a function
```

(suite sur la page suivante)

(suite de la page précédente)

```
# at any time.
my_lock = Lock()

def with_lock[**P, R](f: Callable[Concatenate[Lock, P], R]) -> Callable[P, R]:
    '''A type-safe decorator which provides a lock.'''
    def inner(*args: P.args, **kwargs: P.kwargs) -> R:
        # Provide the lock as the first argument.
        return f(my_lock, *args, **kwargs)
    return inner

@with_lock
def sum_threadsafe(lock: Lock, numbers: list[float]) -> float:
    '''Add a list of numbers together in a thread-safe manner.'''
    with lock:
        return sum(numbers)

# We don't need to pass in the lock ourselves thanks to the decorator.
sum_threadsafe([1.1, 2.2, 3.3])
```

Ajouté dans la version 3.10.

Voir aussi

- **PEP 612** -- Parameter Specification Variables (the PEP which introduced `ParamSpec` and `Concatenate`)
- *`ParamSpec`*
- *Annotating callable objects*

`typing.`**`Literal`**

Special typing form to define "literal types".

`Literal` can be used to indicate to type checkers that the annotated object has a value equivalent to one of the provided literals.

Par exemple :

```
def validate_simple(data: Any) -> Literal[True]:  # always returns True
    ...

type Mode = Literal['r', 'rb', 'w', 'wb']
def open_helper(file: str, mode: Mode) -> str:
    ...

open_helper('/some/path', 'r')      # Passes type check
open_helper('/other/path', 'typo')  # Error in type checker
```

`Literal[...]` ne peut être sous-classé. Lors de l'exécution, une valeur arbitraire est autorisée comme argument de type pour `Literal[...]`, mais les vérificateurs de type peuvent imposer des restrictions. Voir la **PEP 586** pour plus de détails sur les types littéraux.

Ajouté dans la version 3.8.

Modifié dans la version 3.9.1 : `Literal` now de-duplicates parameters. Equality comparisons of `Literal` objects are no longer order dependent. `Literal` objects will now raise a *`TypeError`* exception during equality comparisons if one of their parameters are not *hashable*.

`typing.`**`ClassVar`**

Construction de type particulière pour indiquer les variables de classe.

Telle qu'introduite dans la **PEP 526**, une annotation de variable enveloppée dans ClassVar indique qu'un attribut donné est destiné à être utilisé comme une variable de classe et ne doit pas être défini sur des instances de cette classe. Utilisation :

```
class Starship:
    stats: ClassVar[dict[str, int]] = {} # class variable
    damage: int = 10                     # instance variable
```

`ClassVar` n'accepte que les types et ne peut plus être dérivé.

`ClassVar` n'est pas une classe en soi, et ne devrait pas être utilisée avec `isinstance()` ou `issubclass()`. `ClassVar` ne modifie pas le comportement d'exécution Python, mais il peut être utilisé par des vérificateurs tiers. Par exemple, un vérificateur de types peut marquer le code suivant comme une erreur :

```
enterprise_d = Starship(3000)
enterprise_d.stats = {} # Error, setting class variable on instance
Starship.stats = {}     # This is OK
```

Ajouté dans la version 3.5.3.

Modifié dans la version 3.13 : `ClassVar` can now be nested in `Final` and vice versa.

typing.**Final**

Special typing construct to indicate final names to type checkers.

Final names cannot be reassigned in any scope. Final names declared in class scopes cannot be overridden in subclasses.

Par exemple :

```
MAX_SIZE: Final = 9000
MAX_SIZE += 1  # Error reported by type checker

class Connection:
    TIMEOUT: Final[int] = 10

class FastConnector(Connection):
    TIMEOUT = 1  # Error reported by type checker
```

Ces propriétés ne sont pas vérifiées à l'exécution. Voir la **PEP 591** pour plus de détails.

Ajouté dans la version 3.8.

Modifié dans la version 3.13 : `Final` can now be nested in `ClassVar` and vice versa.

typing.**Required**

Special typing construct to mark a `TypedDict` key as required.

This is mainly useful for `total=False` TypedDicts. See `TypedDict` and **PEP 655** for more details.

Ajouté dans la version 3.11.

typing.**NotRequired**

Special typing construct to mark a `TypedDict` key as potentially missing.

See `TypedDict` and **PEP 655** for more details.

Ajouté dans la version 3.11.

typing.**ReadOnly**

A special typing construct to mark an item of a `TypedDict` as read-only.

Par exemple :

```
class Movie(TypedDict):
   title: ReadOnly[str]
   year: int

def mutate_movie(m: Movie) -> None:
   m["year"] = 1999  # allowed
   m["title"] = "The Matrix"  # typechecker error
```

There is no runtime checking for this property.

See *TypedDict* and **PEP 705** for more details.

Ajouté dans la version 3.13.

typing.**Annotated**

Special typing form to add context-specific metadata to an annotation.

Add metadata `x` to a given type `T` by using the annotation `Annotated[T, x]`. Metadata added using `Annotated` can be used by static analysis tools or at runtime. At runtime, the metadata is stored in a `__metadata__` attribute.

If a library or tool encounters an annotation `Annotated[T, x]` and has no special logic for the metadata, it should ignore the metadata and simply treat the annotation as `T`. As such, `Annotated` can be useful for code that wants to use annotations for purposes outside Python's static typing system.

Using `Annotated[T, x]` as an annotation still allows for static typechecking of `T`, as type checkers will simply ignore the metadata `x`. In this way, `Annotated` differs from the *@no_type_check* decorator, which can also be used for adding annotations outside the scope of the typing system, but completely disables typechecking for a function or class.

The responsibility of how to interpret the metadata lies with the tool or library encountering an `Annotated` annotation. A tool or library encountering an `Annotated` type can scan through the metadata elements to determine if they are of interest (e.g., using *isinstance()*).

Annotated[<type>, <metadata>]

Here is an example of how you might use `Annotated` to add metadata to type annotations if you were doing range analysis :

```
@dataclass
class ValueRange:
    lo: int
    hi: int

T1 = Annotated[int, ValueRange(-10, 5)]
T2 = Annotated[T1, ValueRange(-20, 3)]
```

Details of the syntax :

— The first argument to `Annotated` must be a valid type

— Multiple metadata elements can be supplied (`Annotated` supports variadic arguments) :

```
@dataclass
class ctype:
    kind: str

Annotated[int, ValueRange(3, 10), ctype("char")]
```

It is up to the tool consuming the annotations to decide whether the client is allowed to add multiple metadata elements to one annotation and how to merge those annotations.

— `Annotated` must be subscripted with at least two arguments (`Annotated[int]` is not valid)

— The order of the metadata elements is preserved and matters for equality checks :

```
assert Annotated[int, ValueRange(3, 10), ctype("char")] != Annotated[
    int, ctype("char"), ValueRange(3, 10)
]
```

— Nested `Annotated` types are flattened. The order of the metadata elements starts with the innermost annotation :

```
assert Annotated[Annotated[int, ValueRange(3, 10)], ctype("char")] ==␣
↪Annotated[
    int, ValueRange(3, 10), ctype("char")
]
```

— Duplicated metadata elements are not removed :

```
assert Annotated[int, ValueRange(3, 10)] != Annotated[
    int, ValueRange(3, 10), ValueRange(3, 10)
]
```

— `Annotated` can be used with nested and generic aliases :

```
@dataclass
class MaxLen:
    value: int

type Vec[T] = Annotated[list[tuple[T, T]], MaxLen(10)]

# When used in a type annotation, a type checker will treat "V" the same as
# ``Annotated[list[tuple[int, int]], MaxLen(10)]``:
type V = Vec[int]
```

— `Annotated` cannot be used with an unpacked *`TypeVarTuple`* :

```
type Variadic[*Ts] = Annotated[*Ts, Ann1]  # NOT valid
```

This would be equivalent to :

```
Annotated[T1, T2, T3, ..., Ann1]
```

where `T1`, `T2`, etc. are *`TypeVars`*. This would be invalid : only one type should be passed to Annotated.

— By default, *`get_type_hints()`* strips the metadata from annotations. Pass `include_extras=True` to have the metadata preserved :

```
>>> from typing import Annotated, get_type_hints
>>> def func(x: Annotated[int, "metadata"]) -> None: pass
...
>>> get_type_hints(func)
{'x': <class 'int'>, 'return': <class 'NoneType'>}
>>> get_type_hints(func, include_extras=True)
{'x': typing.Annotated[int, 'metadata'], 'return': <class 'NoneType'>}
```

— At runtime, the metadata associated with an `Annotated` type can be retrieved via the `__metadata__` attribute :

```
>>> from typing import Annotated
>>> X = Annotated[int, "very", "important", "metadata"]
>>> X
typing.Annotated[int, 'very', 'important', 'metadata']
>>> X.__metadata__
('very', 'important', 'metadata')
```

— At runtime, if you want to retrieve the original type wrapped by `Annotated`, use the `__origin__` attribute :

```
>>> from typing import Annotated, get_origin
>>> Password = Annotated[str, "secret"]
>>> Password.__origin__
<class 'str'>
```

Note that using *get_origin()* will return `Annotated` itself :

```
>>> get_origin(Password)
typing.Annotated
```

Voir aussi

PEP 593 - Flexible function and variable annotations
The PEP introducing `Annotated` to the standard library.

Ajouté dans la version 3.9.

`typing.`**`TypeIs`**

Special typing construct for marking user-defined type predicate functions.

`TypeIs` can be used to annotate the return type of a user-defined type predicate function. `TypeIs` only accepts a single type argument. At runtime, functions marked this way should return a boolean and take at least one positional argument.

`TypeIs` aims to benefit *type narrowing* -- a technique used by static type checkers to determine a more precise type of an expression within a program's code flow. Usually type narrowing is done by analyzing conditional code flow and applying the narrowing to a block of code. The conditional expression here is sometimes referred to as a "type predicate" :

```
def is_str(val: str | float):
    # "isinstance" type predicate
    if isinstance(val, str):
        # Type of ``val`` is narrowed to ``str``
        ...
    else:
        # Else, type of ``val`` is narrowed to ``float``.
        ...
```

Sometimes it would be convenient to use a user-defined boolean function as a type predicate. Such a function should use `TypeIs[...]` or *TypeGuard* as its return type to alert static type checkers to this intention. `TypeIs` usually has more intuitive behavior than `TypeGuard`, but it cannot be used when the input and output types are incompatible (e.g., `list[object]` to `list[int]`) or when the function does not return `True` for all instances of the narrowed type.

Using `-> TypeIs[NarrowedType]` tells the static type checker that for a given function :

1. The return value is a boolean.
2. If the return value is `True`, the type of its argument is the intersection of the argument's original type and `NarrowedType`.
3. If the return value is `False`, the type of its argument is narrowed to exclude `NarrowedType`.

Par exemple :

```
from typing import assert_type, final, TypeIs

class Parent: pass
class Child(Parent): pass
@final
class Unrelated: pass
```

(suite sur la page suivante)

(suite de la page précédente)

```
def is_parent(val: object) -> TypeIs[Parent]:
    return isinstance(val, Parent)

def run(arg: Child | Unrelated):
    if is_parent(arg):
        # Type of ``arg`` is narrowed to the intersection
        # of ``Parent`` and ``Child``, which is equivalent to
        # ``Child``.
        assert_type(arg, Child)
    else:
        # Type of ``arg`` is narrowed to exclude ``Parent``,
        # so only ``Unrelated`` is left.
        assert_type(arg, Unrelated)
```

The type inside `TypeIs` must be consistent with the type of the function's argument ; if it is not, static type checkers will raise an error. An incorrectly written `TypeIs` function can lead to unsound behavior in the type system ; it is the user's responsibility to write such functions in a type-safe manner.

If a `TypeIs` function is a class or instance method, then the type in `TypeIs` maps to the type of the second parameter (after `cls` or `self`).

In short, the form `def foo(arg: TypeA) -> TypeIs[TypeB]: ...`, means that if `foo(arg)` returns `True`, then `arg` is an instance of `TypeB`, and if it returns `False`, it is not an instance of `TypeB`.

`TypeIs` also works with type variables. For more information, see **PEP 742** (Narrowing types with `TypeIs`).

Ajouté dans la version 3.13.

`typing.`**`TypeGuard`**

Special typing construct for marking user-defined type predicate functions.

Type predicate functions are user-defined functions that return whether their argument is an instance of a particular type. `TypeGuard` works similarly to *`TypeIs`*, but has subtly different effects on type checking behavior (see below).

Using `-> TypeGuard` tells the static type checker that for a given function :

1. The return value is a boolean.
2. If the return value is `True`, the type of its argument is the type inside `TypeGuard`.

`TypeGuard` also works with type variables. See **PEP 647** for more details.

Par exemple :

```
def is_str_list(val: list[object]) -> TypeGuard[list[str]]:
    '''Determines whether all objects in the list are strings'''
    return all(isinstance(x, str) for x in val)

def func1(val: list[object]):
    if is_str_list(val):
        # Type of ``val`` is narrowed to ``list[str]``.
        print(" ".join(val))
    else:
        # Type of ``val`` remains as ``list[object]``.
        print("Not a list of strings!")
```

`TypeIs` and `TypeGuard` differ in the following ways :

— `TypeIs` requires the narrowed type to be a subtype of the input type, while `TypeGuard` does not. The main reason is to allow for things like narrowing `list[object]` to `list[str]` even though the latter is not a subtype of the former, since `list` is invariant.

— When a `TypeGuard` function returns `True`, type checkers narrow the type of the variable to exactly the `TypeGuard` type. When a `TypeIs` function returns `True`, type checkers can infer a more precise type

combining the previously known type of the variable with the `TypeIs` type. (Technically, this is known as an intersection type.)

— When a `TypeGuard` function returns `False`, type checkers cannot narrow the type of the variable at all. When a `TypeIs` function returns `False`, type checkers can narrow the type of the variable to exclude the `TypeIs` type.

Ajouté dans la version 3.10.

`typing.`**`Unpack`**

Typing operator to conceptually mark an object as having been unpacked.

For example, using the unpack operator `*` on a *type variable tuple* is equivalent to using `Unpack` to mark the type variable tuple as having been unpacked :

```
Ts = TypeVarTuple('Ts')
tup: tuple[*Ts]
# Effectively does:
tup: tuple[Unpack[Ts]]
```

In fact, `Unpack` can be used interchangeably with `*` in the context of *typing.TypeVarTuple* and *builtins.tuple* types. You might see `Unpack` being used explicitly in older versions of Python, where `*` couldn't be used in certain places :

```
# In older versions of Python, TypeVarTuple and Unpack
# are located in the `typing_extensions` backports package.
from typing_extensions import TypeVarTuple, Unpack

Ts = TypeVarTuple('Ts')
tup: tuple[*Ts]         # Syntax error on Python <= 3.10!
tup: tuple[Unpack[Ts]]  # Semantically equivalent, and backwards-compatible
```

`Unpack` can also be used along with *typing.TypedDict* for typing `**kwargs` in a function signature :

```
from typing import TypedDict, Unpack

class Movie(TypedDict):
    name: str
    year: int

# This function expects two keyword arguments - `name` of type `str`
# and `year` of type `int`.
def foo(**kwargs: Unpack[Movie]): ...
```

See **PEP 692** for more details on using `Unpack` for `**kwargs` typing.

Ajouté dans la version 3.11.

Building generic types and type aliases

The following classes should not be used directly as annotations. Their intended purpose is to be building blocks for creating generic types and type aliases.

These objects can be created through special syntax (type parameter lists and the `type` statement). For compatibility with Python 3.11 and earlier, they can also be created without the dedicated syntax, as documented below.

`class` `typing.`**`Generic`**

Classe de base abstraite pour les types génériques.

A generic type is typically declared by adding a list of type parameters after the class name :

```
class Mapping[KT, VT]:
    def __getitem__(self, key: KT) -> VT:
```

(suite sur la page suivante)

(suite de la page précédente)

```
        ...
        # Etc.
```

Such a class implicitly inherits from `Generic`. The runtime semantics of this syntax are discussed in the Language Reference.

Cette classe peut alors être utilisée comme suit :

```
def lookup_name[X, Y](mapping: Mapping[X, Y], key: X, default: Y) -> Y:
    try:
        return mapping[key]
    except KeyError:
        return default
```

Here the brackets after the function name indicate a generic function.

For backwards compatibility, generic classes can also be declared by explicitly inheriting from `Generic`. In this case, the type parameters must be declared separately :

```
KT = TypeVar('KT')
VT = TypeVar('VT')

class Mapping(Generic[KT, VT]):
    def __getitem__(self, key: KT) -> VT:
        ...
        # Etc.
```

class typing.**TypeVar**(*name*, **constraints*, *bound=None*, *covariant=False*, *contravariant=False*, *infer_variance=False*, *default=typing.NoDefault*)

Variables de type.

The preferred way to construct a type variable is via the dedicated syntax for generic functions, generic classes, and generic type aliases :

```
class Sequence[T]:  # T is a TypeVar
    ...
```

This syntax can also be used to create bounded and constrained type variables :

```
class StrSequence[S: str]:  # S is a TypeVar with a `str` upper bound;
    ...                     # we can say that S is "bounded by `str`"


class StrOrBytesSequence[A: (str, bytes)]:  # A is a TypeVar constrained to␣
↪str or bytes
    ...
```

However, if desired, reusable type variables can also be constructed manually, like so :

```
T = TypeVar('T')  # Can be anything
S = TypeVar('S', bound=str)  # Can be any subtype of str
A = TypeVar('A', str, bytes)  # Must be exactly str or bytes
```

Type variables exist primarily for the benefit of static type checkers. They serve as the parameters for generic types as well as for generic function and type alias definitions. See `Generic` for more information on generic types. Generic functions work as follows :

```
def repeat[T](x: T, n: int) -> Sequence[T]:
    """Return a list containing n references to x."""
```

(suite sur la page suivante)

(suite de la page précédente)

```
    return [x]*n


def print_capitalized[S: str](x: S) -> S:
    """Print x capitalized, and return x."""
    print(x.capitalize())
    return x


def concatenate[A: (str, bytes)](x: A, y: A) -> A:
    """Add two strings or bytes objects together."""
    return x + y
```

Note that type variables can be *bounded*, *constrained*, or neither, but cannot be both bounded *and* constrained.

The variance of type variables is inferred by type checkers when they are created through the type parameter syntax or when `infer_variance=True` is passed. Manually created type variables may be explicitly marked covariant or contravariant by passing `covariant=True` or `contravariant=True`. By default, manually created type variables are invariant. See **PEP 484** and **PEP 695** for more details.

Bounded type variables and constrained type variables have different semantics in several important ways. Using a *bounded* type variable means that the `TypeVar` will be solved using the most specific type possible :

```
x = print_capitalized('a string')
reveal_type(x)  # revealed type is str

class StringSubclass(str):
    pass

y = print_capitalized(StringSubclass('another string'))
reveal_type(y)  # revealed type is StringSubclass

z = print_capitalized(45)  # error: int is not a subtype of str
```

The upper bound of a type variable can be a concrete type, abstract type (ABC or Protocol), or even a union of types :

```
# Can be anything with an __abs__ method
def print_abs[T: SupportsAbs](arg: T) -> None:
    print("Absolute value:", abs(arg))

U = TypeVar('U', bound=str|bytes)  # Can be any subtype of the union str|bytes
V = TypeVar('V', bound=SupportsAbs)  # Can be anything with an __abs__ method
```

Using a *constrained* type variable, however, means that the `TypeVar` can only ever be solved as being exactly one of the constraints given :

```
a = concatenate('one', 'two')
reveal_type(a)  # revealed type is str

b = concatenate(StringSubclass('one'), StringSubclass('two'))
reveal_type(b)  # revealed type is str, despite StringSubclass being passed in

c = concatenate('one', b'two')  # error: type variable 'A' can be either str␣
↪or bytes in a function call, but not both
```

At runtime, `isinstance(x, T)` will raise *`TypeError`*.

`__name__`

The name of the type variable.

__covariant__

Whether the type var has been explicitly marked as covariant.

__contravariant__

Whether the type var has been explicitly marked as contravariant.

__infer_variance__

Whether the type variable's variance should be inferred by type checkers.

Ajouté dans la version 3.12.

__bound__

The upper bound of the type variable, if any.

Modifié dans la version 3.12 : For type variables created through type parameter syntax, the bound is evaluated only when the attribute is accessed, not when the type variable is created (see lazy-evaluation).

evaluate_bound()

An *evaluate function* corresponding to the `__bound__` attribute. When called directly, this method supports only the `VALUE` format, which is equivalent to accessing the `__bound__` attribute directly, but the method object can be passed to `annotationlib.call_evaluate_function()` to evaluate the value in a different format.

Ajouté dans la version 3.14.

__constraints__

A tuple containing the constraints of the type variable, if any.

Modifié dans la version 3.12 : For type variables created through type parameter syntax, the constraints are evaluated only when the attribute is accessed, not when the type variable is created (see lazy-evaluation).

evaluate_constraints()

An *evaluate function* corresponding to the `__constraints__` attribute. When called directly, this method supports only the `VALUE` format, which is equivalent to accessing the `__constraints__` attribute directly, but the method object can be passed to `annotationlib.call_evaluate_function()` to evaluate the value in a different format.

Ajouté dans la version 3.14.

__default__

The default value of the type variable, or `typing.NoDefault` if it has no default.

Ajouté dans la version 3.13.

evaluate_default()

An *evaluate function* corresponding to the `__default__` attribute. When called directly, this method supports only the `VALUE` format, which is equivalent to accessing the `__default__` attribute directly, but the method object can be passed to `annotationlib.call_evaluate_function()` to evaluate the value in a different format.

Ajouté dans la version 3.14.

has_default()

Return whether or not the type variable has a default value. This is equivalent to checking whether `__default__` is not the `typing.NoDefault` singleton, except that it does not force evaluation of the lazily evaluated default value.

Ajouté dans la version 3.13.

Modifié dans la version 3.12 : Type variables can now be declared using the type parameter syntax introduced by **PEP 695**. The `infer_variance` parameter was added.

Modifié dans la version 3.13 : Support for default values was added.

class typing.**TypeVarTuple**(*name*, *, *default=typing.NoDefault*)

Type variable tuple. A specialized form of *type variable* that enables *variadic* generics.

Type variable tuples can be declared in type parameter lists using a single asterisk (`*`) before the name :

```
def move_first_element_to_last[T, *Ts](tup: tuple[T, *Ts]) -> tuple[*Ts, T]:
    return (*tup[1:], tup[0])
```

Or by explicitly invoking the `TypeVarTuple` constructor :

```
T = TypeVar("T")
Ts = TypeVarTuple("Ts")

def move_first_element_to_last(tup: tuple[T, *Ts]) -> tuple[*Ts, T]:
    return (*tup[1:], tup[0])
```

A normal type variable enables parameterization with a single type. A type variable tuple, in contrast, allows parameterization with an *arbitrary* number of types by acting like an *arbitrary* number of type variables wrapped in a tuple. For example :

```
# T is bound to int, Ts is bound to ()
# Return value is (1,), which has type tuple[int]
move_first_element_to_last(tup=(1,))

# T is bound to int, Ts is bound to (str,)
# Return value is ('spam', 1), which has type tuple[str, int]
move_first_element_to_last(tup=(1, 'spam'))

# T is bound to int, Ts is bound to (str, float)
# Return value is ('spam', 3.0, 1), which has type tuple[str, float, int]
move_first_element_to_last(tup=(1, 'spam', 3.0))

# This fails to type check (and fails at runtime)
# because tuple[()] is not compatible with tuple[T, *Ts]
# (at least one element is required)
move_first_element_to_last(tup=())
```

Note the use of the unpacking operator `*` in `tuple[T, *Ts]`. Conceptually, you can think of `Ts` as a tuple of type variables `(T1, T2, ...)`. `tuple[T, *Ts]` would then become `tuple[T, *(T1, T2, ...)]`, which is equivalent to `tuple[T, T1, T2, ...]`. (Note that in older versions of Python, you might see this written using *`Unpack`* instead, as `Unpack[Ts]`.)

Type variable tuples must *always* be unpacked. This helps distinguish type variable tuples from normal type variables :

```
x: Ts           # Not valid
x: tuple[Ts]    # Not valid
x: tuple[*Ts]   # The correct way to do it
```

Type variable tuples can be used in the same contexts as normal type variables. For example, in class definitions, arguments, and return types :

```
class Array[*Shape]:
    def __getitem__(self, key: tuple[*Shape]) -> float: ...
    def __abs__(self) -> "Array[*Shape]": ...
    def get_shape(self) -> tuple[*Shape]: ...
```

Type variable tuples can be happily combined with normal type variables :

```
class Array[DType, *Shape]:  # This is fine
    pass

class Array2[*Shape, DType]:  # This would also be fine
    pass

class Height: ...
class Width: ...

float_array_1d: Array[float, Height] = Array()      # Totally fine
int_array_2d: Array[int, Height, Width] = Array()   # Yup, fine too
```

However, note that at most one type variable tuple may appear in a single list of type arguments or type parameters :

```
x: tuple[*Ts, *Ts]             # Not valid
class Array[*Shape, *Shape]:   # Not valid
    pass
```

Finally, an unpacked type variable tuple can be used as the type annotation of `*args` :

```
def call_soon[*Ts](
    callback: Callable[[*Ts], None],
    *args: *Ts
) -> None:
    ...
    callback(*args)
```

In contrast to non-unpacked annotations of `*args` - e.g. `*args: int`, which would specify that *all* arguments are `int` - `*args: *Ts` enables reference to the types of the *individual* arguments in `*args`. Here, this allows us to ensure the types of the `*args` passed to `call_soon` match the types of the (positional) arguments of `callback`.

See **PEP 646** for more details on type variable tuples.

__name__

The name of the type variable tuple.

__default__

The default value of the type variable tuple, or `typing.NoDefault` if it has no default.

Ajouté dans la version 3.13.

evaluate_default()

An *evaluate function* corresponding to the `__default__` attribute. When called directly, this method supports only the `VALUE` format, which is equivalent to accessing the `__default__` attribute directly, but the method object can be passed to `annotationlib.call_evaluate_function()` to evaluate the value in a different format.

Ajouté dans la version 3.14.

has_default()

Return whether or not the type variable tuple has a default value. This is equivalent to checking whether `__default__` is not the `typing.NoDefault` singleton, except that it does not force evaluation of the lazily evaluated default value.

Ajouté dans la version 3.13.

Ajouté dans la version 3.11.

Modifié dans la version 3.12 : Type variable tuples can now be declared using the type parameter syntax introduced by **PEP 695**.

Modifié dans la version 3.13 : Support for default values was added.

class typing.**ParamSpec**(*name*, **, bound=None, covariant=False, contravariant=False, default=typing.NoDefault*)

Parameter specification variable. A specialized version of *type variables*.

In type parameter lists, parameter specifications can be declared with two asterisks (`**`) :

```
type IntFunc[**P] = Callable[P, int]
```

For compatibility with Python 3.11 and earlier, `ParamSpec` objects can also be created as follows :

```
P = ParamSpec('P')
```

Parameter specification variables exist primarily for the benefit of static type checkers. They are used to forward the parameter types of one callable to another callable -- a pattern commonly found in higher order functions and decorators. They are only valid when used in `Concatenate`, or as the first argument to `Callable`, or as parameters for user-defined Generics. See `Generic` for more information on generic types.

For example, to add basic logging to a function, one can create a decorator `add_logging` to log function calls. The parameter specification variable tells the type checker that the callable passed into the decorator and the new callable returned by it have inter-dependent type parameters :

```
from collections.abc import Callable
import logging

def add_logging[T, **P](f: Callable[P, T]) -> Callable[P, T]:
    '''A type-safe decorator to add logging to a function.'''
    def inner(*args: P.args, **kwargs: P.kwargs) -> T:
        logging.info(f'{f.__name__} was called')
        return f(*args, **kwargs)
    return inner

@add_logging
def add_two(x: float, y: float) -> float:
    '''Add two numbers together.'''
    return x + y
```

Without `ParamSpec`, the simplest way to annotate this previously was to use a `TypeVar` with upper bound `Callable[..., Any]`. However this causes two problems :

1. The type checker can't type check the `inner` function because `*args` and `**kwargs` have to be typed `Any`.
2. `cast()` may be required in the body of the `add_logging` decorator when returning the `inner` function, or the static type checker must be told to ignore the `return inner`.

args

kwargs

Since `ParamSpec` captures both positional and keyword parameters, `P.args` and `P.kwargs` can be used to split a `ParamSpec` into its components. `P.args` represents the tuple of positional parameters in a given call and should only be used to annotate `*args`. `P.kwargs` represents the mapping of keyword parameters to their values in a given call, and should be only be used to annotate `**kwargs`. Both attributes require the annotated parameter to be in scope. At runtime, `P.args` and `P.kwargs` are instances respectively of `ParamSpecArgs` and `ParamSpecKwargs`.

__name__

The name of the parameter specification.

__default__

The default value of the parameter specification, or `typing.NoDefault` if it has no default.

Ajouté dans la version 3.13.

evaluate_default()

An *evaluate function* corresponding to the `__default__` attribute. When called directly, this method supports only the `VALUE` format, which is equivalent to accessing the `__default__` attribute directly, but the method object can be passed to `annotationlib.call_evaluate_function()` to evaluate the value in a different format.

Ajouté dans la version 3.14.

has_default()

Return whether or not the parameter specification has a default value. This is equivalent to checking whether `__default__` is not the `typing.NoDefault` singleton, except that it does not force evaluation of the lazily evaluated default value.

Ajouté dans la version 3.13.

Parameter specification variables created with `covariant=True` or `contravariant=True` can be used to declare covariant or contravariant generic types. The `bound` argument is also accepted, similar to `TypeVar`. However the actual semantics of these keywords are yet to be decided.

Ajouté dans la version 3.10.

Modifié dans la version 3.12 : Parameter specifications can now be declared using the type parameter syntax introduced by **PEP 695**.

Modifié dans la version 3.13 : Support for default values was added.

Note

Only parameter specification variables defined in global scope can be pickled.

Voir aussi

- **PEP 612** -- Parameter Specification Variables (the PEP which introduced `ParamSpec` and `Concatenate`)
- `Concatenate`
- *Annotating callable objects*

typing.**ParamSpecArgs**

typing.**ParamSpecKwargs**

Arguments and keyword arguments attributes of a `ParamSpec`. The `P.args` attribute of a `ParamSpec` is an instance of `ParamSpecArgs`, and `P.kwargs` is an instance of `ParamSpecKwargs`. They are intended for runtime introspection and have no special meaning to static type checkers.

Calling `get_origin()` on either of these objects will return the original `ParamSpec` :

```
>>> from typing import ParamSpec, get_origin
>>> P = ParamSpec("P")
>>> get_origin(P.args) is P
True
>>> get_origin(P.kwargs) is P
True
```

Ajouté dans la version 3.10.

class typing.**TypeAliasType**(*name*, *value*, *, *type_params=()*)

The type of type aliases created through the `type` statement.

Example :

```
>>> type Alias = int
>>> type(Alias)
<class 'typing.TypeAliasType'>
```

Ajouté dans la version 3.12.

__name__

The name of the type alias :

```
>>> type Alias = int
>>> Alias.__name__
'Alias'
```

__module__

The module in which the type alias was defined :

```
>>> type Alias = int
>>> Alias.__module__
'__main__'
```

__type_params__

The type parameters of the type alias, or an empty tuple if the alias is not generic :

```
>>> type ListOrSet[T] = list[T] | set[T]
>>> ListOrSet.__type_params__
(T,)
>>> type NotGeneric = int
>>> NotGeneric.__type_params__
()
```

__value__

The type alias's value. This is lazily evaluated, so names used in the definition of the alias are not resolved until the `__value__` attribute is accessed :

```
>>> type Mutually = Recursive
>>> type Recursive = Mutually
>>> Mutually
Mutually
>>> Recursive
Recursive
>>> Mutually.__value__
Recursive
>>> Recursive.__value__
Mutually
```

evaluate_value()

An *evaluate function* corresponding to the `__value__` attribute. When called directly, this method supports only the `VALUE` format, which is equivalent to accessing the `__value__` attribute directly, but the method object can be passed to `annotationlib.call_evaluate_function()` to evaluate the value in a different format :

```
>>> type Alias = undefined
>>> Alias.__value__
Traceback (most recent call last):
...
NameError: name 'undefined' is not defined
>>> from annotationlib import Format, call_evaluate_function
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> Alias.evaluate_value(Format.VALUE)
Traceback (most recent call last):
...
NameError: name 'undefined' is not defined
>>> call_evaluate_function(Alias.evaluate_value, Format.FORWARDREF)
ForwardRef('undefined')
```

Ajouté dans la version 3.14.

Other special directives

These functions and classes should not be used directly as annotations. Their intended purpose is to be building blocks for creating and declaring types.

class typing.**NamedTuple**

Version typée de *collections.namedtuple()*.

Utilisation :

```
class Employee(NamedTuple):
    name: str
    id: int
```

Ce qui est équivalent à :

```
Employee = collections.namedtuple('Employee', ['name', 'id'])
```

Pour assigner une valeur par défaut à un champ, vous pouvez lui donner dans le corps de classe :

```
class Employee(NamedTuple):
    name: str
    id: int = 3

employee = Employee('Guido')
assert employee.id == 3
```

Les champs avec une valeur par défaut doivent venir après tous les champs sans valeur par défaut.

The resulting class has an extra attribute `__annotations__` giving a dict that maps the field names to the field types. (The field names are in the `_fields` attribute and the default values are in the `_field_defaults` attribute, both of which are part of the *namedtuple()* API.)

Les sous-classes de `NamedTuple` peuvent aussi avoir des *docstrings* et des méthodes :

```
class Employee(NamedTuple):
    """Represents an employee."""
    name: str
    id: int = 3

    def __repr__(self) -> str:
        return f'<Employee {self.name}, id={self.id}>'
```

`NamedTuple` subclasses can be generic :

```
class Group[T](NamedTuple):
    key: T
    group: list[T]
```

Utilisation rétrocompatible :

```
# For creating a generic NamedTuple on Python 3.11
T = TypeVar("T")

class Group(NamedTuple, Generic[T]):
    key: T
    group: list[T]

# A functional syntax is also supported
Employee = NamedTuple('Employee', [('name', str), ('id', int)])
```

Modifié dans la version 3.6 : Ajout de la gestion de la syntaxe d'annotation variable de la **PEP 526**.

Modifié dans la version 3.6.1 : Ajout de la prise en charge des valeurs par défaut, des méthodes et des chaînes de caractères *docstrings*.

Modifié dans la version 3.8 : Les attributs `_field_types` et `__annotations__` sont maintenant des dictionnaires standards au lieu d'instances de `OrderedDict`.

Modifié dans la version 3.9 : rend l'attribut `_field_types` obsolète en faveur de l'attribut plus standard `__annotations__` qui a la même information.

Modifié dans la version 3.11 : Added support for generic namedtuples.

Deprecated since version 3.13, will be removed in version 3.15 : The undocumented keyword argument syntax for creating NamedTuple classes (`NT = NamedTuple("NT", x=int)`) is deprecated, and will be disallowed in 3.15. Use the class-based syntax or the functional syntax instead.

Deprecated since version 3.13, will be removed in version 3.15 : When using the functional syntax to create a NamedTuple class, failing to pass a value to the 'fields' parameter (`NT = NamedTuple("NT")`) is deprecated. Passing `None` to the 'fields' parameter (`NT = NamedTuple("NT", None)`) is also deprecated. Both will be disallowed in Python 3.15. To create a NamedTuple class with 0 fields, use `class NT(NamedTuple): pass` or `NT = NamedTuple("NT", [])`.

class typing.**NewType**(*name*, *tp*)

Helper class to create low-overhead *distinct types*.

A `NewType` is considered a distinct type by a typechecker. At runtime, however, calling a `NewType` returns its argument unchanged.

Utilisation :

```
UserId = NewType('UserId', int)  # Declare the NewType "UserId"
first_user = UserId(1)  # "UserId" returns the argument unchanged at runtime
```

__module__

The module in which the new type is defined.

__name__

The name of the new type.

__supertype__

The type that the new type is based on.

Ajouté dans la version 3.5.2.

Modifié dans la version 3.10 : `NewType` is now a class rather than a function.

class typing.**Protocol**(*Generic*)

Base class for protocol classes.

Protocol classes are defined like this :

```
class Proto(Protocol):
    def meth(self) -> int:
```

(suite sur la page suivante)

(suite de la page précédente)

```
        ...
```

Ces classes sont principalement utilisées avec les vérificateurs statiques de type qui reconnaissent les sous-types structurels (typage canard statique), par exemple :

```
class C:
    def meth(self) -> int:
        return 0

def func(x: Proto) -> int:
    return x.meth()

func(C())  # Passes static type check
```

See **PEP 544** for more details. Protocol classes decorated with `runtime_checkable()` (described later) act as simple-minded runtime protocols that check only the presence of given attributes, ignoring their type signatures.

Les classes de protocole peuvent être génériques, par exemple :

```
class GenProto[T](Protocol):
    def meth(self) -> T:
        ...
```

In code that needs to be compatible with Python 3.11 or older, generic Protocols can be written as follows :

```
T = TypeVar("T")

class GenProto(Protocol[T]):
    def meth(self) -> T:
        ...
```

Ajouté dans la version 3.8.

`@typing.runtime_checkable`

Marquez une classe de protocole comme protocole d'exécution.

Such a protocol can be used with `isinstance()` and `issubclass()`. This raises `TypeError` when applied to a non-protocol class. This allows a simple-minded structural check, very similar to "one trick ponies" in `collections.abc` such as `Iterable`. For example :

```
@runtime_checkable
class Closable(Protocol):
    def close(self): ...

assert isinstance(open('/some/file'), Closable)

@runtime_checkable
class Named(Protocol):
    name: str

import threading
assert isinstance(threading.Thread(name='Bob'), Named)
```

Note

`runtime_checkable()` will check only the presence of the required methods or attributes, not their type signatures or types. For example, `ssl.SSLObject` is a class, therefore it passes an `issubclass()` check

against *Callable*. However, the `ssl.SSLObject.__init__` method exists only to raise a `TypeError` with a more informative message, therefore making it impossible to call (instantiate) `ssl.SSLObject`.

Note

An `isinstance()` check against a runtime-checkable protocol can be surprisingly slow compared to an `isinstance()` check against a non-protocol class. Consider using alternative idioms such as `hasattr()` calls for structural checks in performance-sensitive code.

Ajouté dans la version 3.8.

Modifié dans la version 3.12 : The internal implementation of `isinstance()` checks against runtime-checkable protocols now uses `inspect.getattr_static()` to look up attributes (previously, `hasattr()` was used). As a result, some objects which used to be considered instances of a runtime-checkable protocol may no longer be considered instances of that protocol on Python 3.12+, and vice versa. Most users are unlikely to be affected by this change.

Modifié dans la version 3.12 : The members of a runtime-checkable protocol are now considered "frozen" at runtime as soon as the class has been created. Monkey-patching attributes onto a runtime-checkable protocol will still work, but will have no impact on `isinstance()` checks comparing objects to the protocol. See "What's new in Python 3.12" for more details.

class `typing.`**`TypedDict`**(*dict*)

Special construct to add type hints to a dictionary. At runtime it is a plain `dict`.

`TypedDict` declares a dictionary type that expects all of its instances to have a certain set of keys, where each key is associated with a value of a consistent type. This expectation is not checked at runtime but is only enforced by type checkers. Usage :

```
class Point2D(TypedDict):
    x: int
    y: int
    label: str

a: Point2D = {'x': 1, 'y': 2, 'label': 'good'}  # OK
b: Point2D = {'z': 3, 'label': 'bad'}           # Fails type check

assert Point2D(x=1, y=2, label='first') == dict(x=1, y=2, label='first')
```

An alternative way to create a `TypedDict` is by using function-call syntax. The second argument must be a literal `dict` :

```
Point2D = TypedDict('Point2D', {'x': int, 'y': int, 'label': str})
```

This functional syntax allows defining keys which are not valid identifiers, for example because they are keywords or contain hyphens :

```
# raises SyntaxError
class Point2D(TypedDict):
    in: int  # 'in' is a keyword
    x-y: int  # name with hyphens

# OK, functional syntax
Point2D = TypedDict('Point2D', {'in': int, 'x-y': int})
```

By default, all keys must be present in a `TypedDict`. It is possible to mark individual keys as non-required using `NotRequired` :

```
class Point2D(TypedDict):
    x: int
    y: int
    label: NotRequired[str]

# Alternative syntax
Point2D = TypedDict('Point2D', {'x': int, 'y': int, 'label': NotRequired[str]})
```

This means that a `Point2D` `TypedDict` can have the `label` key omitted.

It is also possible to mark all keys as non-required by default by specifying a totality of `False` :

```
class Point2D(TypedDict, total=False):
    x: int
    y: int

# Alternative syntax
Point2D = TypedDict('Point2D', {'x': int, 'y': int}, total=False)
```

This means that a `Point2D` `TypedDict` can have any of the keys omitted. A type checker is only expected to support a literal `False` or `True` as the value of the `total` argument. `True` is the default, and makes all items defined in the class body required.

Individual keys of a `total=False` `TypedDict` can be marked as required using *`Required`* :

```
class Point2D(TypedDict, total=False):
    x: Required[int]
    y: Required[int]
    label: str

# Alternative syntax
Point2D = TypedDict('Point2D', {
    'x': Required[int],
    'y': Required[int],
    'label': str
}, total=False)
```

It is possible for a `TypedDict` type to inherit from one or more other `TypedDict` types using the class-based syntax. Usage :

```
class Point3D(Point2D):
    z: int
```

`Point3D` has three items : `x`, `y` and `z`. It is equivalent to this definition :

```
class Point3D(TypedDict):
    x: int
    y: int
    z: int
```

A `TypedDict` cannot inherit from a non-`TypedDict` class, except for *`Generic`*. For example :

```
class X(TypedDict):
    x: int

class Y(TypedDict):
    y: int

class Z(object): pass  # A non-TypedDict class
```

(suite sur la page suivante)

(suite de la page précédente)

```
class XY(X, Y): pass  # OK

class XZ(X, Z): pass  # raises TypeError
```

A `TypedDict` can be generic :

```
class Group[T](TypedDict):
    key: T
    group: list[T]
```

To create a generic `TypedDict` that is compatible with Python 3.11 or lower, inherit from `Generic` explicitly :

```
T = TypeVar("T")

class Group(TypedDict, Generic[T]):
    key: T
    group: list[T]
```

A `TypedDict` can be introspected via annotations dicts (see annotations-howto for more information on annotations best practices), `__total__`, `__required_keys__`, and `__optional_keys__`.

__total__

`Point2D.__total__` gives the value of the `total` argument. Example :

```
>>> from typing import TypedDict
>>> class Point2D(TypedDict): pass
>>> Point2D.__total__
True
>>> class Point2D(TypedDict, total=False): pass
>>> Point2D.__total__
False
>>> class Point3D(Point2D): pass
>>> Point3D.__total__
True
```

This attribute reflects *only* the value of the `total` argument to the current `TypedDict` class, not whether the class is semantically total. For example, a `TypedDict` with `__total__` set to `True` may have keys marked with `NotRequired`, or it may inherit from another `TypedDict` with `total=False`. Therefore, it is generally better to use `__required_keys__` and `__optional_keys__` for introspection.

__required_keys__

Ajouté dans la version 3.9.

__optional_keys__

`Point2D.__required_keys__` and `Point2D.__optional_keys__` return `frozenset` objects containing required and non-required keys, respectively.

Keys marked with `Required` will always appear in `__required_keys__` and keys marked with `NotRequired` will always appear in `__optional_keys__`.

For backwards compatibility with Python 3.10 and below, it is also possible to use inheritance to declare both required and non-required keys in the same `TypedDict` . This is done by declaring a `TypedDict` with one value for the `total` argument and then inheriting from it in another `TypedDict` with a different value for `total` :

```
>>> class Point2D(TypedDict, total=False):
...     x: int
...     y: int
```

(suite sur la page suivante)

(suite de la page précédente)

```
...
>>> class Point3D(Point2D):
...     z: int
...
>>> Point3D.__required_keys__ == frozenset({'z'})
True
>>> Point3D.__optional_keys__ == frozenset({'x', 'y'})
True
```

Ajouté dans la version 3.9.

Note

If `from __future__ import annotations` is used or if annotations are given as strings, annotations are not evaluated when the `TypedDict` is defined. Therefore, the runtime introspection that `__required_keys__` and `__optional_keys__` rely on may not work properly, and the values of the attributes may be incorrect.

Support for *ReadOnly* is reflected in the following attributes :

__readonly_keys__

A *frozenset* containing the names of all read-only keys. Keys are read-only if they carry the *ReadOnly* qualifier.

Ajouté dans la version 3.13.

__mutable_keys__

A *frozenset* containing the names of all mutable keys. Keys are mutable if they do not carry the *ReadOnly* qualifier.

Ajouté dans la version 3.13.

See **PEP 589** for more examples and detailed rules of using `TypedDict`.

Ajouté dans la version 3.8.

Modifié dans la version 3.11 : Added support for marking individual keys as *Required* or *NotRequired*. See **PEP 655**.

Modifié dans la version 3.11 : Added support for generic `TypedDict`s.

Modifié dans la version 3.13 : Removed support for the keyword-argument method of creating `TypedDict`s.

Modifié dans la version 3.13 : Support for the *ReadOnly* qualifier was added.

Deprecated since version 3.13, will be removed in version 3.15 : When using the functional syntax to create a TypedDict class, failing to pass a value to the 'fields' parameter (`TD = TypedDict("TD")`) is deprecated. Passing `None` to the 'fields' parameter (`TD = TypedDict("TD", None)`) is also deprecated. Both will be disallowed in Python 3.15. To create a TypedDict class with 0 fields, use `class TD(TypedDict): pass` or `TD = TypedDict("TD", {})`.

Protocoles

The following protocols are provided by the typing module. All are decorated with *@runtime_checkable*.

class typing.SupportsAbs

Une ABC avec une méthode abstraite `__abs__` qui est covariante dans son type de retour.

class typing.SupportsBytes

Une ABC avec une méthode abstraite `__bytes__`.

class typing.**SupportsComplex**

Une ABC avec une méthode abstraite `__complex__`.

class typing.**SupportsFloat**

Une ABC avec une méthode abstraite `__float__`.

class typing.**SupportsIndex**

Une ABC avec une méthode abstraite `__index__`.

Ajouté dans la version 3.8.

class typing.**SupportsInt**

Une ABC avec une méthode abstraite `__int__`.

class typing.**SupportsRound**

Une ABC avec une méthode abstraite `__round__` qui est covariante dans son type de retour.

ABCs for working with IO

class typing.**IO**

class typing.**TextIO**

class typing.**BinaryIO**

Generic type `IO[AnyStr]` and its subclasses `TextIO(IO[str])` and `BinaryIO(IO[bytes])` represent the types of I/O streams such as returned by *open()*.

Functions and decorators

typing.**cast**(*typ*, *val*)

Convertit une valeur en un type.

Ceci renvoie la valeur inchangée. Pour le vérificateur de types, cela signifie que la valeur de retour a le type désigné mais, à l'exécution, intentionnellement, rien n'est vérifié (afin que cela soit aussi rapide que possible).

typing.**assert_type**(*val*, *typ*, /)

Vérifie que *val* est bien du type *typ*.

At runtime this does nothing : it returns the first argument unchanged with no checks or side effects, no matter the actual type of the argument.

When a static type checker encounters a call to `assert_type()`, it emits an error if the value is not of the specified type :

```
def greet(name: str) -> None:
    assert_type(name, str)  # OK, inferred type of `name` is `str`
    assert_type(name, int)  # type checker error
```

Cette fonction permet de s'assurer de la compréhension du vérificateur de type d'un script par rapport aux intentions du développeur :

```
def complex_function(arg: object):
    # Do some complex type-narrowing logic,
    # after which we hope the inferred type will be `int`
    ...
    # Test whether the type checker correctly understands our function
    assert_type(arg, int)
```

Ajouté dans la version 3.11.

typing.**assert_never**(*arg*, /)

Demande une confirmation de la part du vérificateur statique de type qu'une ligne de code est inaccessible.

Example :

```
def int_or_str(arg: int | str) -> None:
    match arg:
        case int():
            print("It's an int")
        case str():
            print("It's a str")
        case _ as unreachable:
            assert_never(unreachable)
```

Here, the annotations allow the type checker to infer that the last case can never execute, because `arg` is either an `int` or a `str`, and both options are covered by earlier cases.

If a type checker finds that a call to `assert_never()` is reachable, it will emit an error. For example, if the type annotation for `arg` was instead `int | str | float`, the type checker would emit an error pointing out that `unreachable` is of type `float`. For a call to `assert_never` to pass type checking, the inferred type of the argument passed in must be the bottom type, `Never`, and nothing else.

Une erreur est levé si la fonction est appelé lors de l'exécution.

Voir aussi

Unreachable Code and Exhaustiveness Checking has more information about exhaustiveness checking with static typing.

Ajouté dans la version 3.11.

`typing.`**`reveal_type`**(*obj*, /)

Ask a static type checker to reveal the inferred type of an expression.

When a static type checker encounters a call to this function, it emits a diagnostic with the inferred type of the argument. For example :

```
x: int = 1
reveal_type(x)  # Revealed type is "builtins.int"
```

Cela est utile afin de comprendre comment le vérificateur de types va traiter un bout de code précis.

At runtime, this function prints the runtime type of its argument to `sys.stderr` and returns the argument unchanged (allowing the call to be used within an expression) :

```
x = reveal_type(1)  # prints "Runtime type is int"
print(x)  # prints "1"
```

Note that the runtime type may be different from (more or less specific than) the type statically inferred by a type checker.

Most type checkers support `reveal_type()` anywhere, even if the name is not imported from `typing`. Importing the name from `typing`, however, allows your code to run without runtime errors and communicates intent more clearly.

Ajouté dans la version 3.11.

`@typing.`**`dataclass_transform`**(*, *eq_default=True*, *order_default=False*, *kw_only_default=False*, *frozen_default=False*, *field_specifiers=()*, ***kwargs*)

Decorator to mark an object as providing `dataclass`-like behavior.

`dataclass_transform` may be used to decorate a class, metaclass, or a function that is itself a decorator. The presence of `@dataclass_transform()` tells a static type checker that the decorated object performs runtime "magic" that transforms a class in a similar way to `@dataclasses.dataclass`.

Example usage with a decorator function :

```
@dataclass_transform()
def create_model[T](cls: type[T]) -> type[T]:
    ...
    return cls

@create_model
class CustomerModel:
    id: int
    name: str
```

Avec une classe de base :

```
@dataclass_transform()
class ModelBase: ...

class CustomerModel(ModelBase):
    id: int
    name: str
```

Avec une métaclasse :

```
@dataclass_transform()
class ModelMeta(type): ...

class ModelBase(metaclass=ModelMeta): ...

class CustomerModel(ModelBase):
    id: int
    name: str
```

Les classes `CustomerModel` définis ci-dessus sont traitées par les vérificateurs de type de la même que les classes créées avec *`@dataclasses.dataclass`*. Par exemple, les vérificateurs de type déduisent que ces classes possèdent une méthode `__init__` acceptant `id` et `name` comme arguments.

Les arguments booléens suivants sont acceptés,les vérificateurs de type supposent qu'ils ont le même effet qu'ils auraient eu sur le décorateur *`@dataclasses.dataclass`* : `init`, `eq`, `order`, `unsafe_hash`, `frozen`, `match_args`, `kw_only`, et `slots`. Il est possible d'évaluer statiquement les valeurs de ces arguments (`True` or `False`).

Les arguments du décorateur `dataclass_transform` permettent de personnaliser le comportement par défaut de la classe, métaclasse ou fonction décorée :

Paramètres

- **`eq_default`** (`bool`) -- Indicates whether the `eq` parameter is assumed to be `True` or `False` if it is omitted by the caller. Defaults to `True`.
- **`order_default`** (`bool`) -- Indicates whether the `order` parameter is assumed to be `True` or `False` if it is omitted by the caller. Defaults to `False`.
- **`kw_only_default`** (`bool`) -- Indicates whether the `kw_only` parameter is assumed to be `True` or `False` if it is omitted by the caller. Defaults to `False`.
- **`frozen_default`** (`bool`) -- Indicates whether the `frozen` parameter is assumed to be `True` or `False` if it is omitted by the caller. Defaults to `False`.

  Ajouté dans la version 3.12.
- **`field_specifiers`** (`tuple[Callable[..., Any], ...]`) -- Specifies a static list of supported classes or functions that describe fields, similar to *`dataclasses.field()`*. Defaults to `()`.
- **`**kwargs`** (*`Any`*) -- D'autres arguments sont acceptés afin d'autoriser de futurs possibles extensions.

Type checkers recognize the following optional parameters on field specifiers :

TABLEAU 1 – **Recognised parameters for field specifiers**

Parameter name	Description
`init`	Indicates whether the field should be included in the synthesized `__init__` method. If unspecified, `init` defaults to `True`.
`default`	Provides the default value for the field.
`default_factory`	Provides a runtime callback that returns the default value for the field. If neither `default` nor `default_factory` are specified, the field is assumed to have no default value and must be provided a value when the class is instantiated.
`factory`	An alias for the `default_factory` parameter on field specifiers.
`kw_only`	Indicates whether the field should be marked as keyword-only. If `True`, the field will be keyword-only. If `False`, it will not be keyword-only. If unspecified, the value of the `kw_only` parameter on the object decorated with `dataclass_transform` will be used, or if that is unspecified, the value of `kw_only_default` on `dataclass_transform` will be used.
`alias`	Provides an alternative name for the field. This alternative name is used in the synthesized `__init__` method.

Lors de l'exécution, les arguments de ce décorateur sont enregistrés au sein de l'attribut `__dataclass_transform__` de l'objet décoré. Il n'y pas d'autre effet à l'exécution.

See **PEP 681** for more details.

Ajouté dans la version 3.11.

`@typing.`**`overload`**

Decorator for creating overloaded functions and methods.

The `@overload` decorator allows describing functions and methods that support multiple different combinations of argument types. A series of `@overload`-decorated definitions must be followed by exactly one non-`@overload`-decorated definition (for the same function/method).

`@overload`-decorated definitions are for the benefit of the type checker only, since they will be overwritten by the non-`@overload`-decorated definition. The non-`@overload`-decorated definition, meanwhile, will be used at runtime but should be ignored by a type checker. At runtime, calling an `@overload`-decorated function directly will raise *`NotImplementedError`*.

An example of overload that gives a more precise type than can be expressed using a union or a type variable :

```
@overload
def process(response: None) -> None:
    ...
@overload
def process(response: int) -> tuple[int, str]:
    ...
@overload
def process(response: bytes) -> str:
    ...
def process(response):
    ...  # actual implementation goes here
```

See **PEP 484** for more details and comparison with other typing semantics.

Modifié dans la version 3.11 : Les fonctions surchargées peuvent maintenant être inspectées durant l'exécution via *`get_overloads()`*.

`typing.`**`get_overloads`**(*func*)

Return a sequence of *`@overload`*-decorated definitions for *func*.

func is the function object for the implementation of the overloaded function. For example, given the definition of `process` in the documentation for *`@overload`*, `get_overloads(process)` will return a se-

quence of three function objects for the three defined overloads. If called on a function with no overloads, `get_overloads()` returns an empty sequence.

`get_overloads()` peut être utilisé afin d'inspecter une fonction surchargée durant l'exécution.

Ajouté dans la version 3.11.

`typing.clear_overloads()`

Clear all registered overloads in the internal registry.

This can be used to reclaim the memory used by the registry.

Ajouté dans la version 3.11.

`@typing.final`

Decorator to indicate final methods and final classes.

Decorating a method with `@final` indicates to a type checker that the method cannot be overridden in a subclass. Decorating a class with `@final` indicates that it cannot be subclassed.

Par exemple :

```
class Base:
    @final
    def done(self) -> None:
        ...
class Sub(Base):
    def done(self) -> None:  # Error reported by type checker
        ...

@final
class Leaf:
    ...
class Other(Leaf):  # Error reported by type checker
    ...
```

Ces propriétés ne sont pas vérifiées à l'exécution. Voir la **PEP 591** pour plus de détails.

Ajouté dans la version 3.8.

Modifié dans la version 3.11 : The decorator will now attempt to set a `__final__` attribute to `True` on the decorated object. Thus, a check like `if getattr(obj, "__final__", False)` can be used at runtime to determine whether an object `obj` has been marked as final. If the decorated object does not support setting attributes, the decorator returns the object unchanged without raising an exception.

`@typing.no_type_check`

Décorateur pour indiquer que les annotations ne sont pas des indications de type.

This works as a class or function *decorator*. With a class, it applies recursively to all methods and classes defined in that class (but not to methods defined in its superclasses or subclasses). Type checkers will ignore all annotations in a function or class with this decorator.

`@no_type_check` mutates the decorated object in place.

`@typing.no_type_check_decorator`

Décorateur pour donner à un autre décorateur l'effet *`no_type_check()`*.

Ceci enveloppe le décorateur avec quelque chose qui enveloppe la fonction décorée dans *`no_type_check()`*.

Deprecated since version 3.13, will be removed in version 3.15 : No type checker ever added support for `@no_type_check_decorator`. It is therefore deprecated, and will be removed in Python 3.15.

`@typing.override`

Decorator to indicate that a method in a subclass is intended to override a method or attribute in a superclass.

Type checkers should emit an error if a method decorated with `@override` does not, in fact, override anything.

This helps prevent bugs that may occur when a base class is changed without an equivalent change to a child class.

For example :

```
class Base:
    def log_status(self) -> None:
        ...

class Sub(Base):
    @override
    def log_status(self) -> None:  # Okay: overrides Base.log_status
        ...

    @override
    def done(self) -> None:  # Error reported by type checker
        ...
```

There is no runtime checking of this property.

The decorator will attempt to set an `__override__` attribute to `True` on the decorated object. Thus, a check like `if getattr(obj, "__override__", False)` can be used at runtime to determine whether an object `obj` has been marked as an override. If the decorated object does not support setting attributes, the decorator returns the object unchanged without raising an exception.

See **PEP 698** for more details.

Ajouté dans la version 3.12.

`@typing.type_check_only`

Decorator to mark a class or function as unavailable at runtime.

Ce décorateur n'est pas disponible à l'exécution. Il est principalement destiné à marquer les classes qui sont définies dans des fichiers séparés d'annotations de type (*type stub file*, en anglais) si une implémentation renvoie une instance d'une classe privée :

```
@type_check_only
class Response:  # private or not available at runtime
    code: int
    def get_header(self, name: str) -> str: ...

def fetch_response() -> Response: ...
```

Notez qu'il n'est pas recommandé de renvoyer les instances des classes privées. Il est généralement préférable de rendre ces classes publiques.

Utilitaires d'introspection

`typing.get_type_hints(obj, globalns=None, localns=None, include_extras=False)`

Renvoie un dictionnaire contenant des annotations de type pour une fonction, une méthode, un module ou un objet de classe.

This is often the same as `obj.__annotations__`, but this function makes the following changes to the annotations dictionary :

- Forward references encoded as string literals or `ForwardRef` objects are handled by evaluating them in *globalns*, *localns*, and (where applicable) *obj*'s type parameter namespace. If *globalns* or *localns* is not given, appropriate namespace dictionaries are inferred from *obj*.
- `None` is replaced with `types.NoneType`.
- If `@no_type_check` has been applied to *obj*, an empty dictionary is returned.
- If *obj* is a class `C`, the function returns a dictionary that merges annotations from `C`'s base classes with those on `C` directly. This is done by traversing `C.__mro__` and iteratively combining `__annotations__` dictionaries. Annotations on classes appearing earlier in the *method resolution order* always take precedence

over annotations on classes appearing later in the method resolution order.

— The function recursively replaces all occurrences of `Annotated[T, ...]` with `T`, unless *include_extras* is set to `True` (see `Annotated` for more information).

See also `inspect.get_annotations()`, a lower-level function that returns annotations more directly.

> **Note**
>
> If any forward references in the annotations of *obj* are not resolvable or are not valid Python code, this function will raise an exception such as `NameError`. For example, this can happen with imported *type aliases* that include forward references, or with names imported under `if TYPE_CHECKING`.

Modifié dans la version 3.9 : Added `include_extras` parameter as part of **PEP 593**. See the documentation on `Annotated` for more information.

Modifié dans la version 3.11 : Avant, `Optional[t]` était ajouté pour les annotations de fonctions et de méthodes dans le cas où une valeur par défaut était égal à `None`. Maintenant, les annotations sont renvoyées inchangées.

`typing.`**`get_origin`**(*tp*)

Get the unsubscripted version of a type : for a typing object of the form `X[Y, Z, ...]` return `X`.

If `X` is a typing-module alias for a builtin or `collections` class, it will be normalized to the original class. If `X` is an instance of `ParamSpecArgs` or `ParamSpecKwargs`, return the underlying `ParamSpec`. Return `None` for unsupported objects.

Examples :

```
assert get_origin(str) is None
assert get_origin(Dict[str, int]) is dict
assert get_origin(Union[int, str]) is Union
assert get_origin(Annotated[str, "metadata"]) is Annotated
P = ParamSpec('P')
assert get_origin(P.args) is P
assert get_origin(P.kwargs) is P
```

Ajouté dans la version 3.8.

`typing.`**`get_args`**(*tp*)

Get type arguments with all substitutions performed : for a typing object of the form `X[Y, Z, ...]` return `(Y, Z, ...)`.

If `X` is a union or `Literal` contained in another generic type, the order of `(Y, Z, ...)` may be different from the order of the original arguments `[Y, Z, ...]` due to type caching. Return `()` for unsupported objects.

Examples :

```
assert get_args(int) == ()
assert get_args(Dict[int, str]) == (int, str)
assert get_args(Union[int, str]) == (int, str)
```

Ajouté dans la version 3.8.

`typing.`**`get_protocol_members`**(*tp*)

Return the set of members defined in a `Protocol`.

```
>>> from typing import Protocol, get_protocol_members
>>> class P(Protocol):
...     def a(self) -> str: ...
...     b: int
>>> get_protocol_members(P) == frozenset({'a', 'b'})
True
```

Raise *TypeError* for arguments that are not Protocols.

Ajouté dans la version 3.13.

typing.**is_protocol**(*tp*)

Determine if a type is a *Protocol*.

Par exemple :

```
class P(Protocol):
    def a(self) -> str: ...
    b: int

is_protocol(P)     # => True
is_protocol(int)   # => False
```

Ajouté dans la version 3.13.

typing.**is_typeddict**(*tp*)

Vérifier si un type est un *TypedDict*.

For example :

```
class Film(TypedDict):
    title: str
    year: int

assert is_typeddict(Film)
assert not is_typeddict(list | str)

# TypedDict is a factory for creating typed dicts,
# not a typed dict itself
assert not is_typeddict(TypedDict)
```

Ajouté dans la version 3.10.

class typing.**ForwardRef**

Class used for internal typing representation of string forward references.

For example, `List["SomeClass"]` is implicitly transformed into `List[ForwardRef("SomeClass")]`. `ForwardRef` should not be instantiated by a user, but may be used by introspection tools.

> **Note**
>
> Les types **PEP 585** tels que `list["SomeClass"]` ne seront pas implicitement transformés en `list[ForwardRef("SomeClass")]` et ne seront donc pas automatiquement résolus en `list[SomeClass]`.

Ajouté dans la version 3.7.4.

Modifié dans la version 3.14 : This is now an alias for *annotationlib.ForwardRef*.

typing.**evaluate_forward_ref**(*forward_ref*, *, *owner=None*, *globals=None*, *locals=None*, *type_params=None*, *format=annotationlib.Format.VALUE*)

Evaluate an *annotationlib.ForwardRef* as a *type hint*.

This is similar to calling *annotationlib.ForwardRef.evaluate()*, but unlike that method, `evaluate_forward_ref()` also :

— Recursively evaluates forward references nested within the type hint.
— Raises *TypeError* when it encounters certain objects that are not valid type hints.
— Replaces type hints that evaluate to `None` with *types.NoneType*.
— Supports the *FORWARDREF* and *STRING* formats.

forward_ref must be an instance of `ForwardRef`. *owner*, if given, should be the object that holds the annotations that the forward reference derived from, such as a module, class object, or function. It is used to infer the namespaces to use for looking up names. *globals* and *locals* can also be explicitly given to provide the global and local namespaces. *type_params* is a tuple of type parameters that are in scope when evaluating the forward reference. This parameter must be provided (though it may be an empty tuple) if *owner* is not given and the forward reference does not already have an owner set. *format* specifies the format of the annotation and is a member of the `annotationlib.Format` enum.

Ajouté dans la version 3.14.

typing.**NoDefault**

A sentinel object used to indicate that a type parameter has no default value. For example :

```
>>> T = TypeVar("T")
>>> T.__default__ is typing.NoDefault
True
>>> S = TypeVar("S", default=None)
>>> S.__default__ is None
True
```

Ajouté dans la version 3.13.

Constante

typing.**TYPE_CHECKING**

A special constant that is assumed to be `True` by 3rd party static type checkers. It is `False` at runtime.

Utilisation :

```
if TYPE_CHECKING:
    import expensive_mod

def fun(arg: 'expensive_mod.SomeType') -> None:
    local_var: expensive_mod.AnotherType = other_fun()
```

The first type annotation must be enclosed in quotes, making it a "forward reference", to hide the `expensive_mod` reference from the interpreter runtime. Type annotations for local variables are not evaluated, so the second annotation does not need to be enclosed in quotes.

> **Note**
>
> Si `from __future__ import annotations` est utilisé, les annotations ne sont pas évaluées au moment de la définition de fonction. Elles sont alors stockées comme des chaînes de caractères dans `__annotations__`, ce qui rend inutile l'utilisation de guillemets autour de l'annotation (Voir **PEP 563**).

Ajouté dans la version 3.5.2.

Deprecated aliases

This module defines several deprecated aliases to pre-existing standard library classes. These were originally included in the typing module in order to support parameterizing these generic classes using `[]`. However, the aliases became redundant in Python 3.9 when the corresponding pre-existing classes were enhanced to support `[]` (see **PEP 585**).

The redundant types are deprecated as of Python 3.9. However, while the aliases may be removed at some point, removal of these aliases is not currently planned. As such, no deprecation warnings are currently issued by the interpreter for these aliases.

If at some point it is decided to remove these deprecated aliases, a deprecation warning will be issued by the interpreter for at least two releases prior to removal. The aliases are guaranteed to remain in the typing module without deprecation warnings until at least Python 3.14.

Type checkers are encouraged to flag uses of the deprecated types if the program they are checking targets a minimum Python version of 3.9 or newer.

Aliases to built-in types

class typing.**Dict**(*dict, MutableMapping[KT, VT]*)

Deprecated alias to `dict`.

Note that to annotate arguments, it is preferred to use an abstract collection type such as `Mapping` rather than to use `dict` or `typing.Dict`.

Obsolète depuis la version 3.9 : `builtins.dict` now supports subscripting (`[]`). See **PEP 585** and *Type Alias générique*.

class typing.**List**(*list, MutableSequence[T]*)

Deprecated alias to `list`.

Note that to annotate arguments, it is preferred to use an abstract collection type such as `Sequence` or `Iterable` rather than to use `list` or `typing.List`.

Obsolète depuis la version 3.9 : `builtins.list` now supports subscripting (`[]`). See **PEP 585** and *Type Alias générique*.

class typing.**Set**(*set, MutableSet[T]*)

Deprecated alias to `builtins.set`.

Note that to annotate arguments, it is preferred to use an abstract collection type such as `collections.abc.Set` rather than to use `set` or `typing.Set`.

Obsolète depuis la version 3.9 : `builtins.set` now supports subscripting (`[]`). See **PEP 585** and *Type Alias générique*.

class typing.**FrozenSet**(*frozenset, AbstractSet[T_co]*)

Deprecated alias to `builtins.frozenset`.

Obsolète depuis la version 3.9 : `builtins.frozenset` now supports subscripting (`[]`). See **PEP 585** and *Type Alias générique*.

typing.**Tuple**

Deprecated alias for `tuple`.

`tuple` and `Tuple` are special-cased in the type system ; see *Annotating tuples* for more details.

Obsolète depuis la version 3.9 : `builtins.tuple` now supports subscripting (`[]`). See **PEP 585** and *Type Alias générique*.

class typing.**Type**(*Generic[CT_co]*)

Deprecated alias to `type`.

See *The type of class objects* for details on using `type` or `typing.Type` in type annotations.

Ajouté dans la version 3.5.2.

Obsolète depuis la version 3.9 : `builtins.type` now supports subscripting (`[]`). See **PEP 585** and *Type Alias générique*.

Aliases to types in `collections`

class typing.**DefaultDict**(*collections.defaultdict, MutableMapping[KT, VT]*)

Deprecated alias to `collections.defaultdict`.

Ajouté dans la version 3.5.2.

Obsolète depuis la version 3.9 : `collections.defaultdict` now supports subscripting (`[]`). See **PEP 585** and *Type Alias générique*.

class typing.**OrderedDict**(*collections.OrderedDict, MutableMapping[KT, VT]*)

Deprecated alias to `collections.OrderedDict`.

Ajouté dans la version 3.7.2.

Obsolète depuis la version 3.9 : `collections.OrderedDict` now supports subscripting (`[]`). See **PEP 585** and *Type Alias générique*.

class typing.**ChainMap**(*collections.ChainMap, MutableMapping[KT, VT]*)

Deprecated alias to `collections.ChainMap`.

Ajouté dans la version 3.6.1.

Obsolète depuis la version 3.9 : `collections.ChainMap` now supports subscripting (`[]`). See **PEP 585** and *Type Alias générique*.

class typing.**Counter**(*collections.Counter, Dict[T, int]*)

Deprecated alias to `collections.Counter`.

Ajouté dans la version 3.6.1.

Obsolète depuis la version 3.9 : `collections.Counter` now supports subscripting (`[]`). See **PEP 585** and *Type Alias générique*.

class typing.**Deque**(*deque, MutableSequence[T]*)

Deprecated alias to `collections.deque`.

Ajouté dans la version 3.6.1.

Obsolète depuis la version 3.9 : `collections.deque` now supports subscripting (`[]`). See **PEP 585** and *Type Alias générique*.

Aliases to other concrete types

class typing.**Pattern**

class typing.**Match**

Deprecated aliases corresponding to the return types from `re.compile()` and `re.match()`.

These types (and the corresponding functions) are generic over `AnyStr`. `Pattern` can be specialised as `Pattern[str]` or `Pattern[bytes]` ; `Match` can be specialised as `Match[str]` or `Match[bytes]`.

Obsolète depuis la version 3.9 : Classes `Pattern` and `Match` from `re` now support `[]`. See **PEP 585** and *Type Alias générique*.

class typing.**Text**

Deprecated alias for `str`.

`Text` is provided to supply a forward compatible path for Python 2 code : in Python 2, `Text` is an alias for `unicode`.

Utilisez `Text` pour indiquer qu'une valeur doit contenir une chaîne Unicode d'une manière compatible avec Python 2 et Python 3 :

```
def add_unicode_checkmark(text: Text) -> Text:
    return text + u' \u2713'
```

Ajouté dans la version 3.5.2.

Obsolète depuis la version 3.11 : Python 2 is no longer supported, and most type checkers also no longer support type checking Python 2 code. Removal of the alias is not currently planned, but users are encouraged to use `str` instead of `Text`.

Aliases to container ABCs in `collections.abc`

class `typing.`**`AbstractSet`**(*Collection[T_co]*)

Deprecated alias to `collections.abc.Set`.

Obsolète depuis la version 3.9 : `collections.abc.Set` now supports subscripting (`[]`). See **PEP 585** and *Type Alias générique*.

class `typing.`**`Collection`**(*Sized, Iterable[T_co], Container[T_co]*)

Deprecated alias to `collections.abc.Collection`.

Ajouté dans la version 3.6.

Obsolète depuis la version 3.9 : `collections.abc.Collection` now supports subscripting (`[]`). See **PEP 585** and *Type Alias générique*.

class `typing.`**`Container`**(*Generic[T_co]*)

Deprecated alias to `collections.abc.Container`.

Obsolète depuis la version 3.9 : `collections.abc.Container` now supports subscripting (`[]`). See **PEP 585** and *Type Alias générique*.

class `typing.`**`ItemsView`**(*MappingView, AbstractSet[tuple[KT_co, VT_co]]*)

Deprecated alias to `collections.abc.ItemsView`.

Obsolète depuis la version 3.9 : `collections.abc.ItemsView` now supports subscripting (`[]`). See **PEP 585** and *Type Alias générique*.

class `typing.`**`KeysView`**(*MappingView, AbstractSet[KT_co]*)

Deprecated alias to `collections.abc.KeysView`.

Obsolète depuis la version 3.9 : `collections.abc.KeysView` now supports subscripting (`[]`). See **PEP 585** and *Type Alias générique*.

class `typing.`**`Mapping`**(*Collection[KT], Generic[KT, VT_co]*)

Deprecated alias to `collections.abc.Mapping`.

Obsolète depuis la version 3.9 : `collections.abc.Mapping` now supports subscripting (`[]`). See **PEP 585** and *Type Alias générique*.

class `typing.`**`MappingView`**(*Sized*)

Deprecated alias to `collections.abc.MappingView`.

Obsolète depuis la version 3.9 : `collections.abc.MappingView` now supports subscripting (`[]`). See **PEP 585** and *Type Alias générique*.

class `typing.`**`MutableMapping`**(*Mapping[KT, VT]*)

Deprecated alias to `collections.abc.MutableMapping`.

Obsolète depuis la version 3.9 : `collections.abc.MutableMapping` now supports subscripting (`[]`). See **PEP 585** and *Type Alias générique*.

class `typing.`**`MutableSequence`**(*Sequence[T]*)

Deprecated alias to `collections.abc.MutableSequence`.

Obsolète depuis la version 3.9 : `collections.abc.MutableSequence` now supports subscripting (`[]`). See **PEP 585** and *Type Alias générique*.

class `typing.`**`MutableSet`**(*AbstractSet[T]*)

Deprecated alias to `collections.abc.MutableSet`.

Obsolète depuis la version 3.9 : `collections.abc.MutableSet` now supports subscripting (`[]`). See **PEP 585** and *Type Alias générique*.

class `typing.`**`Sequence`**(*Reversible[T_co], Collection[T_co]*)

Deprecated alias to `collections.abc.Sequence`.

Obsolète depuis la version 3.9 : `collections.abc.Sequence` now supports subscripting (`[]`). See **PEP 585** and *Type Alias générique*.

class typing.**ValuesView**(*MappingView, Collection[_VT_co]*)

Deprecated alias to `collections.abc.ValuesView`.

Obsolète depuis la version 3.9 : `collections.abc.ValuesView` now supports subscripting (`[]`). See **PEP 585** and *Type Alias générique*.

Aliases to asynchronous ABCs in `collections.abc`

class typing.**Coroutine**(*Awaitable[ReturnType], Generic[YieldType, SendType, ReturnType]*)

Deprecated alias to `collections.abc.Coroutine`.

See *Annotating generators and coroutines* for details on using `collections.abc.Coroutine` and `typing.Coroutine` in type annotations.

Ajouté dans la version 3.5.3.

Obsolète depuis la version 3.9 : `collections.abc.Coroutine` now supports subscripting (`[]`). See **PEP 585** and *Type Alias générique*.

class typing.**AsyncGenerator**(*AsyncIterator[YieldType], Generic[YieldType, SendType]*)

Deprecated alias to `collections.abc.AsyncGenerator`.

See *Annotating generators and coroutines* for details on using `collections.abc.AsyncGenerator` and `typing.AsyncGenerator` in type annotations.

Ajouté dans la version 3.6.1.

Obsolète depuis la version 3.9 : `collections.abc.AsyncGenerator` now supports subscripting (`[]`). See **PEP 585** and *Type Alias générique*.

Modifié dans la version 3.13 : The `SendType` parameter now has a default.

class typing.**AsyncIterable**(*Generic[T_co]*)

Deprecated alias to `collections.abc.AsyncIterable`.

Ajouté dans la version 3.5.2.

Obsolète depuis la version 3.9 : `collections.abc.AsyncIterable` now supports subscripting (`[]`). See **PEP 585** and *Type Alias générique*.

class typing.**AsyncIterator**(*AsyncIterable[T_co]*)

Deprecated alias to `collections.abc.AsyncIterator`.

Ajouté dans la version 3.5.2.

Obsolète depuis la version 3.9 : `collections.abc.AsyncIterator` now supports subscripting (`[]`). See **PEP 585** and *Type Alias générique*.

class typing.**Awaitable**(*Generic[T_co]*)

Deprecated alias to `collections.abc.Awaitable`.

Ajouté dans la version 3.5.2.

Obsolète depuis la version 3.9 : `collections.abc.Awaitable` now supports subscripting (`[]`). See **PEP 585** and *Type Alias générique*.

Aliases to other ABCs in `collections.abc`

class typing.**Iterable**(*Generic[T_co]*)

Deprecated alias to `collections.abc.Iterable`.

Obsolète depuis la version 3.9 : `collections.abc.Iterable` now supports subscripting (`[]`). See **PEP 585** and *Type Alias générique*.

class typing.**Iterator**(*Iterable[T_co]*)

Deprecated alias to `collections.abc.Iterator`.

Obsolète depuis la version 3.9 : `collections.abc.Iterator` now supports subscripting (`[]`). See **PEP 585** and *Type Alias générique*.

typing.**Callable**

Deprecated alias to `collections.abc.Callable`.

See *Annotating callable objects* for details on how to use `collections.abc.Callable` and `typing.Callable` in type annotations.

Obsolète depuis la version 3.9 : `collections.abc.Callable` now supports subscripting (`[]`). See **PEP 585** and *Type Alias générique*.

Modifié dans la version 3.10 : `Callable` prend désormais en charge `ParamSpec` et `Concatenate`. Voir **PEP 612** pour plus de détails.

class typing.**Generator**(*Iterator[YieldType], Generic[YieldType, SendType, ReturnType]*)

Deprecated alias to `collections.abc.Generator`.

See *Annotating generators and coroutines* for details on using `collections.abc.Generator` and `typing.Generator` in type annotations.

Obsolète depuis la version 3.9 : `collections.abc.Generator` now supports subscripting (`[]`). See **PEP 585** and *Type Alias générique*.

Modifié dans la version 3.13 : Default values for the send and return types were added.

class typing.**Hashable**

Deprecated alias to `collections.abc.Hashable`.

Obsolète depuis la version 3.12 : Use `collections.abc.Hashable` directly instead.

class typing.**Reversible**(*Iterable[T_co]*)

Deprecated alias to `collections.abc.Reversible`.

Obsolète depuis la version 3.9 : `collections.abc.Reversible` now supports subscripting (`[]`). See **PEP 585** and *Type Alias générique*.

class typing.**Sized**

Deprecated alias to `collections.abc.Sized`.

Obsolète depuis la version 3.12 : Use `collections.abc.Sized` directly instead.

Aliases to `contextlib` ABCs

class typing.**ContextManager**(*Generic[T_co, ExitT_co]*)

Deprecated alias to `contextlib.AbstractContextManager`.

The first type parameter, `T_co`, represents the type returned by the `__enter__()` method. The optional second type parameter, `ExitT_co`, which defaults to `bool | None`, represents the type returned by the `__exit__()` method.

Ajouté dans la version 3.5.4.

Obsolète depuis la version 3.9 : `contextlib.AbstractContextManager` now supports subscripting (`[]`). See **PEP 585** and *Type Alias générique*.

Modifié dans la version 3.13 : Added the optional second type parameter, `ExitT_co`.

class typing.**AsyncContextManager**(*Generic[T_co, AExitT_co]*)

Deprecated alias to `contextlib.AbstractAsyncContextManager`.

The first type parameter, `T_co`, represents the type returned by the `__aenter__()` method. The optional second type parameter, `AExitT_co`, which defaults to `bool | None`, represents the type returned by the `__aexit__()` method.

Ajouté dans la version 3.6.2.

Obsolète depuis la version 3.9 : `contextlib.AbstractAsyncContextManager` now supports subscripting (`[]`). See **PEP 585** and *Type Alias générique*.

Modifié dans la version 3.13 : Added the optional second type parameter, `AExitT_co`.

27.1.13 Étapes d'Obsolescence des Fonctionnalités Majeures

Certaines fonctionnalités dans `typing` sont obsolètes et peuvent être supprimées dans une future version de Python. Le tableau suivant résume les principales dépréciations. Celui-ci peut changer et toutes les dépréciations ne sont pas listées.

Fonctionnalité	Obsolète en	Suppression prévue	PEP/issue
Versions de typage des collections standards	3.9	Undecided (see *Deprecated aliases* for more information)	**PEP 585**
`typing.Text`	3.11	Non défini	gh-92332
`typing.Hashable` and `typing.Sized`	3.12	Non défini	gh-94309
`typing.TypeAlias`	3.12	Non défini	**PEP 695**
`@typing.no_type_check_decorator`	3.13	3.15	gh-106309
`typing.AnyStr`	3.13	3.18	gh-105578

27.2 `pydoc` --- Documentation generator and online help system

Code source : Lib/pydoc.py

The `pydoc` module automatically generates documentation from Python modules. The documentation can be presented as pages of text on the console, served to a web browser, or saved to HTML files.

For modules, classes, functions and methods, the displayed documentation is derived from the docstring (i.e. the `__doc__` attribute) of the object, and recursively of its documentable members. If there is no docstring, `pydoc` tries to obtain a description from the block of comment lines just above the definition of the class, function or method in the source file, or at the top of the module (see `inspect.getcomments()`).

The built-in function `help()` invokes the online help system in the interactive interpreter, which uses `pydoc` to generate its documentation as text on the console. The same text documentation can also be viewed from outside the Python interpreter by running **pydoc** as a script at the operating system's command prompt. For example, running

```
python -m pydoc sys
```

dans un terminal, cela affiche la documentation du module `sys` dans un style similaire à la commande Unix **man**. On peut passer comme argument à **pydoc** le nom d'une fonction, d'un module, d'un paquet, ou une référence pointant vers une classe, une méthode, ou une fonction dans un module ou dans un paquet. Si l'argument passé à **pydoc** est un chemin (c.-à-d. qu'il contient des séparateurs de chemin tels que la barre oblique `/` dans Unix), et fait référence à un fichier source Python existant, alors la documentation est générée pour ce fichier.

> **Note**
>
> In order to find objects and their documentation, `pydoc` imports the module(s) to be documented. Therefore, any code on module level will be executed on that occasion. Use an `if __name__ == '__main__':` guard to only execute code when a file is invoked as a script and not just imported.

When printing output to the console, **pydoc** attempts to paginate the output for easier reading. If either the `MANPAGER` or the `PAGER` environment variable is set, **pydoc** will use its value as a pagination program. When both are set, `MANPAGER` is used.

Ajouter une option `-w` avant l'argument entraine l'enregistrement de la documentation HTML générée dans un fichier du répertoire courant au lieu de l'afficher dans la console.

Ajouter une option `-w` avant l'argument cherche les lignes de résumé de tous les modules disponibles pour le mot clé donné comme argument, ceci à la manière de la commande Unix **man**. Les lignes de résumé d'un module sont les premières lignes de sa *docstring*.

You can also use **pydoc** to start an HTTP server on the local machine that will serve documentation to visiting web browsers. **python -m pydoc -p 1234** will start a HTTP server on port 1234, allowing you to browse the documentation at `http://localhost:1234/` in your preferred web browser. Specifying `0` as the port number will select an arbitrary unused port.

python -m pydoc -n <hostname> will start the server listening at the given hostname. By default the hostname is 'localhost' but if you want the server to be reached from other machines, you may want to change the host name that the server responds to. During development this is especially useful if you want to run pydoc from within a container.

python -m pydoc -b will start the server and additionally open a web browser to a module index page. Each served page has a navigation bar at the top where you can *Get* help on an individual item, *Search* all modules with a keyword in their synopsis line, and go to the *Module index*, *Topics* and *Keywords* pages.

Quand **pydoc** génère de la documentation, il utilise l'environnement et le chemin courant pour localiser les modules. Ainsi, en invoquant les documents **pydoc spam** en précisant la version du module, vous obtenez le même résultat qu'en lançant l'interpréteur Python et en tapant la commande `import spam`.

Module docs for core modules are assumed to reside in `https://docs.python.org/X.Y/library/` where `X` and `Y` are the major and minor version numbers of the Python interpreter. This can be overridden by setting the `PYTHONDOCS` environment variable to a different URL or to a local directory containing the Library Reference Manual pages.

Modifié dans la version 3.2 : Ajout de l'option `-b`.

Modifié dans la version 3.3 : Suppression de l'option `-g`.

Modifié dans la version 3.4 : `pydoc` now uses *`inspect.signature()`* rather than *`inspect.getfullargspec()`* to extract signature information from callables.

Modifié dans la version 3.7 : Ajout de l'option `-n`.

27.3 Python Development Mode

Ajouté dans la version 3.7.

The Python Development Mode introduces additional runtime checks that are too expensive to be enabled by default. It should not be more verbose than the default if the code is correct ; new warnings are only emitted when an issue is detected.

It can be enabled using the `-X dev` command line option or by setting the `PYTHONDEVMODE` environment variable to `1`.

See also Python debug build.

27.3.1 Effects of the Python Development Mode

Enabling the Python Development Mode is similar to the following command, but with additional effects described below :

```
PYTHONMALLOC=debug PYTHONASYNCIODEBUG=1 python -W default -X faulthandler
```

Effects of the Python Development Mode :

- Add `default` *warning filter*. The following warnings are shown :
  - *`DeprecationWarning`*

  - *ImportWarning*
  - *PendingDeprecationWarning*
  - *ResourceWarning*

  Normally, the above warnings are filtered by the default *warning filters*.

  It behaves as if the `-W default` command line option is used.

  Use the `-W error` command line option or set the `PYTHONWARNINGS` environment variable to `error` to treat warnings as errors.
- Install debug hooks on memory allocators to check for :
  - Buffer underflow
  - Buffer overflow
  - Memory allocator API violation
  - Unsafe usage of the GIL

  See the `PyMem_SetupDebugHooks()` C function.

  It behaves as if the `PYTHONMALLOC` environment variable is set to `debug`.

  To enable the Python Development Mode without installing debug hooks on memory allocators, set the `PYTHONMALLOC` environment variable to `default`.
- Call *faulthandler.enable()* at Python startup to install handlers for the *SIGSEGV*, *SIGFPE*, *SIGABRT*, *SIGBUS* and *SIGILL* signals to dump the Python traceback on a crash.

  It behaves as if the `-X faulthandler` command line option is used or if the `PYTHONFAULTHANDLER` environment variable is set to `1`.
- Enable *asyncio debug mode*. For example, *asyncio* checks for coroutines that were not awaited and logs them.

  It behaves as if the `PYTHONASYNCIODEBUG` environment variable is set to `1`.
- Check the *encoding* and *errors* arguments for string encoding and decoding operations. Examples : *open()*, *str.encode()* and *bytes.decode()*.

  By default, for best performance, the *errors* argument is only checked at the first encoding/decoding error and the *encoding* argument is sometimes ignored for empty strings.
- The *io.IOBase* destructor logs `close()` exceptions.
- Set the *dev_mode* attribute of *sys.flags* to `True`.

The Python Development Mode does not enable the *tracemalloc* module by default, because the overhead cost (to performance and memory) would be too large. Enabling the *tracemalloc* module provides additional information on the origin of some errors. For example, *ResourceWarning* logs the traceback where the resource was allocated, and a buffer overflow error logs the traceback where the memory block was allocated.

The Python Development Mode does not prevent the `-O` command line option from removing `assert` statements nor from setting *__debug__* to `False`.

The Python Development Mode can only be enabled at the Python startup. Its value can be read from *sys.flags.dev_mode*.

Modifié dans la version 3.8 : The *io.IOBase* destructor now logs `close()` exceptions.

Modifié dans la version 3.9 : The *encoding* and *errors* arguments are now checked for string encoding and decoding operations.

27.3.2 ResourceWarning Example

Example of a script counting the number of lines of the text file specified in the command line :

```
import sys

def main():
    fp = open(sys.argv[1])
    nlines = len(fp.readlines())
    print(nlines)
    # The file is closed implicitly
```

(suite sur la page suivante)

(suite de la page précédente)

```
if __name__ == "__main__":
    main()
```

The script does not close the file explicitly. By default, Python does not emit any warning. Example using README.txt, which has 269 lines :

```
$ python script.py README.txt
269
```

Enabling the Python Development Mode displays a `ResourceWarning` warning :

```
$ python -X dev script.py README.txt
269
script.py:10: ResourceWarning: unclosed file <_io.TextIOWrapper name='README.rst'␣
↪mode='r' encoding='UTF-8'>
  main()
ResourceWarning: Enable tracemalloc to get the object allocation traceback
```

In addition, enabling `tracemalloc` shows the line where the file was opened :

```
$ python -X dev -X tracemalloc=5 script.py README.rst
269
script.py:10: ResourceWarning: unclosed file <_io.TextIOWrapper name='README.rst'␣
↪mode='r' encoding='UTF-8'>
  main()
Object allocated at (most recent call last):
  File "script.py", lineno 10
    main()
  File "script.py", lineno 4
    fp = open(sys.argv[1])
```

The fix is to close explicitly the file. Example using a context manager :

```
def main():
    # Close the file explicitly when exiting the with block
    with open(sys.argv[1]) as fp:
        nlines = len(fp.readlines())
    print(nlines)
```

Not closing a resource explicitly can leave a resource open for way longer than expected ; it can cause severe issues upon exiting Python. It is bad in CPython, but it is even worse in PyPy. Closing resources explicitly makes an application more deterministic and more reliable.

27.3.3 Bad file descriptor error example

Script displaying the first line of itself :

```
import os

def main():
    fp = open(__file__)
    firstline = fp.readline()
    print(firstline.rstrip())
    os.close(fp.fileno())
    # The file is closed implicitly

main()
```

By default, Python does not emit any warning :

```
$ python script.py
import os
```

The Python Development Mode shows a `ResourceWarning` and logs a "Bad file descriptor" error when finalizing the file object :

```
$ python -X dev script.py
import os
script.py:10: ResourceWarning: unclosed file <_io.TextIOWrapper name='script.py'␣
↪mode='r' encoding='UTF-8'>
  main()
ResourceWarning: Enable tracemalloc to get the object allocation traceback
Exception ignored in: <_io.TextIOWrapper name='script.py' mode='r' encoding='UTF-8
↪'>
Traceback (most recent call last):
  File "script.py", line 10, in <module>
    main()
OSError: [Errno 9] Bad file descriptor
```

`os.close(fp.fileno())` closes the file descriptor. When the file object finalizer tries to close the file descriptor again, it fails with the `Bad file descriptor` error. A file descriptor must be closed only once. In the worst case scenario, closing it twice can lead to a crash (see bpo-18748 for an example).

The fix is to remove the `os.close(fp.fileno())` line, or open the file with `closefd=False`.

27.4 `doctest` — Exemples de tests interactifs en Python

Code source : Lib/doctest.py

Le module `doctest` cherche des extraits de texte ressemblant à des sessions Python interactives avant de les exécuter, de façon à vérifier que le fonctionnement correspond exactement à la description. Voici quelques cas d'utilisation de *doctest* :

- Vérifier que les *docstrings* d'un module sont à jour en vérifiant que tous les exemples interactifs fonctionnent toujours tels que décrits.
- Réaliser un test de régression en vérifiant que les exemples interactifs provenant d'un fichier de test ou d'un objet de test fonctionnent comme prévu.
- Rédiger de la documentation sous forme de tutoriel pour un paquet, avec une abondance d'exemples ayant des entrées et des sorties. On pourrait voir ça comme des tests « dans le texte » ou de la « documentation exécutable », selon le point de vue.

Voici un petit exemple d'un module qui soit tout de même complet :

```
"""
This is the "example" module.

The example module supplies one function, factorial().  For example,

>>> factorial(5)
120
"""

def factorial(n):
    """Return the factorial of n, an exact integer >= 0.

    >>> [factorial(n) for n in range(6)]
    [1, 1, 2, 6, 24, 120]
```

(suite sur la page suivante)

(suite de la page précédente)

```
    >>> factorial(30)
    265252859812191058636308480000000
    >>> factorial(-1)
    Traceback (most recent call last):
        ...
    ValueError: n must be >= 0

    Factorials of floats are OK, but the float must be an exact integer:
    >>> factorial(30.1)
    Traceback (most recent call last):
        ...
    ValueError: n must be exact integer
    >>> factorial(30.0)
    265252859812191058636308480000000

    It must also not be ridiculously large:
    >>> factorial(1e100)
    Traceback (most recent call last):
        ...
    OverflowError: n too large
    """

    import math
    if not n >= 0:
        raise ValueError("n must be >= 0")
    if math.floor(n) != n:
        raise ValueError("n must be exact integer")
    if n+1 == n:  # catch a value like 1e300
        raise OverflowError("n too large")
    result = 1
    factor = 2
    while factor <= n:
        result *= factor
        factor += 1
    return result


if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

L'exécution du fichier `example.py` directement à partir de la ligne de commande démontre la magie de *doctest* :

```
$ python example.py
$
```

Il n'y a pas de sortie ! C'est normal, cela signifie que tous les exemples fonctionnent. Passez `-v` au script pour que *doctest* affiche un journal détaillé de ce qui a été essayé, avant d'afficher un résumé à la fin :

```
$ python example.py -v
Trying:
    factorial(5)
Expecting:
    120
ok
Trying:
    [factorial(n) for n in range(6)]
```

(suite sur la page suivante)

(suite de la page précédente)

```
Expecting:
    [1, 1, 2, 6, 24, 120]
ok
```

Et ainsi de suite, jusqu'à ce qu'on atteigne :

```
Trying:
    factorial(1e100)
Expecting:
    Traceback (most recent call last):
        ...
    OverflowError: n too large
ok
2 items passed all tests:
   1 test in __main__
   6 tests in __main__.factorial
7 tests in 2 items.
7 passed.
Test passed.
$
```

That's all you need to know to start making productive use of `doctest`! Jump in. The following sections provide full details. Note that there are many examples of doctests in the standard Python test suite and libraries. Especially useful examples can be found in the standard test file `Lib/test/test_doctest/test_doctest.py`.

Ajouté dans la version 3.13 : Output is colorized by default and can be controlled using environment variables.

27.4.1 Utilisation simple : vérifier des exemples dans des *docstrings*

Le plus simple pour commencer à utiliser *doctest* (mais pas nécessairement la façon avec laquelle vous continuerez) est de terminer chaque module `M` avec :

```
if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

`doctest` examine alors les *docstrings* dans le module `M`.

Exécuter le module comme un script a comme conséquence d'exécuter et de vérifier les exemples dans les *docstrings* :

```
python M.py
```

Ceci n'affiche rien à moins qu'un exemple échoue ; le cas échéant, les exemples défaillants et les causes du ou des échecs sont affichés sur *stdout*, et la ligne finale de la sortie est `***Test Failed*** *N failures*.`, où *N* est le nombre d'exemples défaillants.

À la place, exécutez-la avec l'option de ligne de commande `-v` :

```
python M.py -v
```

alors, un rapport détaillé de tous les exemples faisant partie de l'essai est affiché sur la sortie standard, accompagné à la fin de leurs résumés.

Vous pouvez activer le mode verbeux en passant `verbose=True` à `testmod()`, ou vous le désactiver en lui passant `verbose=False`. Dans ces deux cas, `sys.argv` n'est pas inspecté par `testmod()` (ainsi, lui passer `-v` ou pas n'a aucun effet).

Il y a un raccourci pour exécuter `testmod()` à partir de la ligne de commande. Vous demandez à l'interpréteur Python d'exécuter le module *doctest* directement à partir de la bibliothèque standard afin de passer le ou les noms des modules à partir de la ligne de commande ainsi :

```
python -m doctest -v example.py
```

Ceci importera `example.py` comme un module autonome et exécutera `testmod()` sur celui-ci. Notez que ceci peut ne pas fonctionner si le fichier fait partie d'un paquet et importe d'autres sous-modules de ce paquet.

Pour plus d'informations sur `testmod()`, consultez la section *API de base*.

27.4.2 Utilisation simple : vérifier des exemples dans un fichier texte

Une autre application simple de *doctest* est de tester des exemples interactifs dans un fichier texte. Ceci est fait avec la fonction `testfile()` :

```
import doctest
doctest.testfile("example.txt")
```

Ce court script exécute et vérifie chacun des exemples Python interactifs contenus dans le fichier `example.txt`. Le contenu du fichier est traité comme une seule *docstring* géante ; le fichier n'a pas besoin de contenir un programme Python ! Par exemple, prenons un fichier `example.txt` contenant :

```
The ``example`` module
======================

Using ``factorial``
-------------------

This is an example text file in reStructuredText format.  First import
``factorial`` from the ``example`` module:

    >>> from example import factorial

Now use it:

    >>> factorial(6)
    120
```

Exécuter `doctest.testfile("example.txt")` recherche les erreurs dans cette documentation :

```
File "./example.txt", line 14, in example.txt
Failed example:
    factorial(6)
Expected:
    120
Got:
    720
```

Comme pour `testmod()`, `testfile()` n'affichera rien sauf si un exemple échoue. Si un exemple échoue, alors le ou les exemples défaillants et leurs causes sont affichés sur *stdout*, dans le même format que `testmod()`.

Par défaut, `testfile()` cherche les fichiers dans le répertoire où se situe le module qui l'appelle. Consultez la section *API de base* pour une description des options de ligne de commande à utiliser afin de chercher dans d'autres répertoires.

Comme pour `testmod()`, la verbosité de `testfile()` peut être ajustée avec l'option de ligne de commande `-v` ou avec le mot clé *verbose*.

Il y a un raccourci pour exécuter `testfile()` à partir de la ligne de commande. Demandez à l'interpréteur Python d'exécuter le module *doctest* directement à partir de la bibliothèque standard et de passer le ou les noms des modules à partir de la ligne de commande ainsi :

```
python -m doctest -v example.txt
```

Puisque le nom du fichier ne se termine pas par `.py`, `doctest` en déduit qu'il s'exécute à l'aide de `testfile()`, et non pas `testmod()`.

Pour plus d'information sur `testfile()`, consultez la section *API de base*.

27.4.3 Comment ça marche

Cette section examine en détail le fonctionnement de *doctest* : quelles *docstrings* sont considérées, comment sont trouvés les exemples interactifs, quel est le contexte d'exécution sélectionné, comment les exceptions sont gérées, et de quelles façons les options de ligne de commande peuvent être utilisées pour définir le comportement. Ceci est l'information dont vous avez besoin pour écrire des exemples *doctest* ; pour de l'information sur l'exécution de *doctest* sur ces exemples, consultez les sections suivantes.

Quelles *docstrings* sont considérées ?

Les *docstrings* du module, de toutes les fonctions, classes, et méthodes sont cherchées. Les objets qui sont importés dans le module ne sont pas cherchés.

In addition, there are cases when you want tests to be part of a module but not part of the help text, which requires that the tests not be included in the docstring. Doctest looks for a module-level variable called `__test__` and uses it to locate other tests. If `M.__test__` exists, it must be a dict, and each entry maps a (string) name to a function object, class object, or string. Function and class object docstrings found from `M.__test__` are searched, and strings are treated as if they were docstrings. In output, a key `K` in `M.__test__` appears with name `M.__test__.K`.

For example, place this block of code at the top of `example.py` :

```
__test__ = {
    'numbers': """
>>> factorial(6)
720

>>> [factorial(n) for n in range(6)]
[1, 1, 2, 6, 24, 120]
"""
}
```

The value of `example.__test__["numbers"]` will be treated as a docstring and all the tests inside it will be run. It is important to note that the value can be mapped to a function, class object, or module ; if so, `doctest` searches them recursively for docstrings, which are then scanned for tests.

Toute classe trouvée est ainsi cherchée récursivement, afin de tester les *docstrings* contenues dans leurs méthodes et leurs classes imbriquées.

Comment les exemples *docstring* sont-ils identifiés ?

Dans la plupart des cas, un copier-coller d'une séance interactive de console fonctionne bien, mais *doctest* n'essaye pas de faire une simulation exacte d'un *shell* Python spécifique.

```
>>> # comments are ignored
>>> x = 12
>>> x
12
>>> if x == 13:
...     print("yes")
... else:
...     print("no")
...     print("NO")
...     print("NO!!!")
```

(suite sur la page suivante)

(suite de la page précédente)

```
...
no
NO
NO!!!
>>>
```

Toute sortie souhaitée doit immédiatement suivre le dernier `'>>> '` ou le dernier `'... '` contenant le code, et la sortie souhaitée, s'il y en a une, s'étend jusqu'au prochain `'>>> '` ou à la prochaine ligne vide.

En détail :

- La sortie souhaitée ne peut pas contenir une ligne vide, puisque contenir une telle ligne signale la fin de la sortie souhaitée. Si la sortie souhaitée doit contenir une ligne vide, ajoutez `<BLANKLINE>` dans votre exemple *doctest* à chaque endroit où une ligne vide est souhaitée.
- Tous les caractères de tabulation insécables (*hard tab characters*) sont convertis en espaces, en utilisant des taquets de tabulation de 8 espaces. Les tabulations se trouvant dans la sortie générée par le code test ne sont pas modifiées. Comme tout caractère de tabulation insécable *est* converti, ceci veut dire que si le code de sortie inclut des caractères de tabulation insécables, alors la seule façon que le *doctest* peut réussir est si l'option `NORMALIZE_WHITESPACE` ou si *directive* a cours. De façon alternative, le test peut être ré-écrit afin de capturer la sortie et de la comparer à un ensemble de valeurs attendues, et ce, en tant qu'étape du test. Cette gestion des tabulations à la source a été obtenue suite à un processus d'essais et d'erreurs ; il a été démontré que c'était là la façon de les gérer qui soit la moins susceptible de générer des erreurs. Il est possible d'utiliser un algorithme différent pour la gestion des tabulations en rédigeant une classe sur mesure `DocTestParser`.
- La sortie vers *stdout* est capturée, mais pas la sortie vers *stderr* (les traces d'appel sont capturées par d'autres moyens).
- Si vous souhaitez conserver les barres obliques inversées telles quelles lorsque vous terminez une ligne avec une barre oblique inversée dans une séance interactive, ou quand vous utilisez une telle barre pour toute autre raison, vous devez utiliser une *docstring* brute :

  ```
  >>> def f(x):
  ...     r'''Backslashes in a raw docstring: m\n'''
  ...
  >>> print(f.__doc__)
  Backslashes in a raw docstring: m\n
  ```

  Sinon, la barre oblique inversée est interprétée comme faisant partie de la chaîne de caractères. Par exemple, le `\n` ci-dessus est interprété comme un caractère de saut de ligne. De façon alternative, vous pouvez doubler chaque barre oblique inversée dans la version *doctest* (et n'utilisez pas dans ce cas de *docstring* brute) :

  ```
  >>> def f(x):
  ...     '''Backslashes in a raw docstring: m\\n'''
  ...
  >>> print(f.__doc__)
  Backslashes in a raw docstring: m\n
  ```

- La colonne de départ n'a pas d'importance :

  ```
  >>> assert "Easy!"
        >>> import math
            >>> math.floor(1.9)
            1
  ```

  et autant d'espaces sont retirés de la sortie attendue qu'il y avait d'espaces avant la ligne commençant par `'>>> '`.

Quel est le contexte d'exécution ?

Par défaut, chaque fois que `doctest` trouve une *docstring* à tester, il utilise une *copie superficielle* des variables globales de `M`, de telle façon que l'exécution de tests ne change pas les variables globales réelles du module et que l'exécution d'un unique test dans `M` ne puisse laisser traîner des miettes pouvant accidentellement causer la réussite d'un autre test. Ceci signifie que les exemples peuvent utiliser librement n'importe quel nom défini au niveau supérieur dans `M` ainsi que les noms définis précédemment dans la *docstring* en cours d'exécution. Les exemples ne peuvent voir les noms définis dans d'autres *docstrings*.

Vous pouvez forcer l'utilisation de votre propre *dict* comme contexte d'exécution en passant `globs=your_dict` à `testmod()` ou encore, à `testfile()`.

Qu'en est-il des exceptions ?

Pas de problèmes, tant que la trace d'appels est la seule sortie produite par l'exemple : il suffit d'ajouter la trace.[1] Comme les traces d'appels contiennent des détails qui sont sujets à changement rapide (par exemple, le chemin exact vers un fichier et les numéros de ligne), ceci est un cas où *doctest* fait un effort pour être flexible dans ce qu'il accepte.

Exemple simple :

```
>>> [1, 2, 3].remove(42)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
```

Ce *doctest* réussit si `ValueError` est levée, avec le détail `list.remote(x): x not in list` tel que montré.

La sortie attendue pour une exception doit commencer par un en-tête de trace d'appels, qui peut être l'une des deux lignes suivantes, avec la même indentation que la première ligne de l'exemple :

```
Traceback (most recent call last):
Traceback (innermost last):
```

L'en-tête de la trace d'appels est suivi par une pile optionnelle de trace d'appels, dont le contenu est ignoré par *doctest*. La trace d'appels est habituellement omise, ou est copiée verbatim à partir d'une séance interactive.

La pile de trace d'appels est suivie par la partie la plus intéressante : la ou les lignes contenant le type et le détail de l'exception. Ceci est habituellement la dernière ligne de la trace d'appels ; dans le cas où l'exception a un détail sur plusieurs lignes, il est possible de prolonger sur plusieurs lignes :

```
>>> raise ValueError('multi\n    line\ndetail')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: multi
    line
detail
```

Les trois dernières lignes (en commençant par `ValueError`) sont comparées avec le type et le détail de l'exception ; tout le reste est ignoré.

La pratique optimale est d'omettre la pile de trace d'appels, à moins que celle-ci ait une valeur significative de documentation de l'exemple. Ainsi, le dernier exemple est probablement meilleur tel qui suit :

```
>>> raise ValueError('multi\n    line\ndetail')
Traceback (most recent call last):
    ...
ValueError: multi
    line
detail
```

1. Les exemples contenant à la fois la sortie attendue et une exception ne sont pas supportés. Tenter de deviner où finit l'un et où commence l'autre peut mener à plusieurs erreurs, en plus d'être un test qui soit source de confusion.

Prenez note que les traces d'appels sont traitées de façon vraiment particulière. Précisément, dans l'exemple ré-écrit, l'utilisation de `...` est indépendante de l'option *doctest* `ELLIPSIS`. Les points de suspension dans cet exemple peuvent être omis, ou peuvent aussi être trois (ou trois cents) virgules ou chiffres, ou une retranscription indentée d'une parodie de Monty Python.

Quelques détails que vous devriez lire une fois, mais que vous pouvez oublier :

- *Doctest* ne peut pas deviner si votre sortie attendue provient d'une trace d'appels issue d'une exception ou d'un affichage ordinaire. Ainsi, si nous avons un exemple s'attendant à obtenir `ValueError: 42 is prime`, celui-ci réussira peu importe si `ValueError` est réellement levée ou si l'exemple affiche simplement ce texte de trace d'appels. Dans la pratique, une sortie ordinaire commence rarement par une ligne d'en-tête de trace d'appels ; ainsi, ceci ne pose pas de vrai problème.
- Chaque ligne de la trace d'appel (s'il y en a) doit soit être indentée d'un niveau supplémentaire au niveau de la première ligne de l'exemple *ou* doit commencer par un caractère qui ne soit pas alphanumérique. La première ligne suivant l'en-tête de la trace d'appels qui soit indentée similairement et qui commence par un caractère alphanumérique est comprise comme étant le début du détail de l'exception. Bien sûr, ceci fait la chose adéquate pour les traces d'appels véritables.
- Lorsque l'option de *doctest* `IGNORE_EXCEPTION_DETAIL` est définie, tout ce qui suit le point-virgule se trouvant le plus à gauche ainsi que toute information liée au module dans le nom de l'exception sont ignorés.
- Le shell interactif omet la ligne d'en-tête de la trace d'appels pour certaines erreurs `SyntaxError`. Ceci étant dit, *doctest* utilise la ligne d'en-tête de la trace d'appels afin de faire une distinction entre les exceptions et les autres types d'erreurs. Ainsi, dans les rares cas où vous avez besoin de tester une erreur `SyntaxError` qui omet l'en-tête de la trace d'appels, il vous est nécessaire d'ajouter manuellement la ligne d'en-tête de la trace d'appels à l'exemple de test.
- Pour certaines erreurs, Python affiche la position de l'erreur en utilisant le marqueur `^` et des tildes :

```
>>> 1 + None
  File "<stdin>", line 1
    1 + None
    ~~^~~~~~
TypeError: unsupported operand type(s) for +: 'int' and 'NoneType'
```

Comme les lignes dénotant la position de l'erreur précèdent le type et le détail de l'exception, elles ne sont pas vérifiées par *doctest*. Par exemple, le test suivant réussira, même si le marqueur `^` n'est pas à la bonne place :

```
>>> 1 + None
  File "<stdin>", line 1
    1 + None
    ^~~~~~~
TypeError: unsupported operand type(s) for +: 'int' and 'NoneType'
```

Options de ligne de commande

Un ensemble d'options de ligne de commande contrôle différents aspects du comportement de *doctest*. Pour les options, des noms symboliques sont fournis comme des constantes de module, qui peuvent être composés par un OU bit à bit (bitwise ORed) et passés à diverses fonctions. Les noms peuvent aussi être utilisés dans des *instructions doctest*, et peuvent être passés à l'interface de ligne de commande de *doctest* à l'aide de l'option `-o`.

Ajouté dans la version 3.4 : L'option de ligne de commande `-o`.

Le premier groupe d'options définit les sémantiques de test, de façon à contrôler comment *doctest* décide si la sortie obtenue correspond à la sortie attendue de l'exemple :

`doctest.`**`DONT_ACCEPT_TRUE_FOR_1`**

Par défaut, si un bloc de sortie attendu contient uniquement un `1`, un vrai bloc de sortie contenant uniquement un `1` ou un `True` sera considéré comme étant une correspondance ; de façon similaire, nous avons une correspondance pour `0` et `False`. Lorsque l'option `DONT_ACCEPT_TRUE_FOR_1` est précisée, aucune de ces substitutions n'est acceptée. Le comportement par défaut s'ajuste au fait que Python a changé le type de renvoi de plusieurs fonctions, passant de nombres entiers à des booléens ; les *doctests* s'attendant à une sortie de « petit entier » (*little integer*) fonctionnent encore dans ces cas. Cette option disparaîtra probablement, mais pas avant plusieurs années.

doctest.**DONT_ACCEPT_BLANKLINE**

Par défaut, si un bloc de sortie attendue contient une ligne contenant uniquement la chaîne de caractères `<BLANKLINE>`, alors cette ligne sera en correspondance avec une ligne vide dans la sortie réelle. Puisqu'une véritable ligne vide permet de délimiter la sortie attendue, ceci est la seule façon de communiquer qu'une ligne vide est souhaitée. Lorsque l'option *DONT_ACCEPT_BLANKLINE* est précisée, cette substitution n'est pas permise.

doctest.**NORMALIZE_WHITESPACE**

Lorsque précisé, toutes les séquences de caractères d'espacement et de caractères de saut de ligne sont traitées comme équivalentes. Toute séquence de caractères d'espacement à l'intérieur de la sortie attendue correspondra alors à toute séquence de caractères d'espacement à l'intérieur de la sortie réelle. Par défaut, les caractères d'espacement doivent correspondre de façon exacte. L'option *NORMALIZE_WHITESPACE* est particulièrement utile lorsqu'une ligne de sortie attendue est très longue, et que l'on souhaite la répartir sur plusieurs lignes dans le fichier source.

doctest.**ELLIPSIS**

Lorsque précisé, un marqueur de points de suspension (`...`) dans la sortie attendue peut correspondre à n'importe quelle partie de chaîne de caractères dans la sortie réelle. Ceci inclut les parties qui traversent les frontières de lignes, ainsi que les parties vides de chaînes de caractères ; ainsi, il est préférable d'en faire une utilisation simple. Les usages complexes mènent aux mêmes surprises du type "oups, il y avait trop de correspondances !" que l'utilisation de `.*` dans les expressions régulières.

doctest.**IGNORE_EXCEPTION_DETAIL**

When specified, doctests expecting exceptions pass so long as an exception of the expected type is raised, even if the details (message and fully qualified exception name) don't match.

For example, an example expecting `ValueError: 42` will pass if the actual exception raised is `ValueError: 3*14`, but will fail if, say, a *TypeError* is raised instead. It will also ignore any fully qualified name included before the exception class, which can vary between implementations and versions of Python and the code/libraries in use. Hence, all three of these variations will work with the flag specified :

```
>>> raise Exception('message')
Traceback (most recent call last):
Exception: message

>>> raise Exception('message')
Traceback (most recent call last):
builtins.Exception: message

>>> raise Exception('message')
Traceback (most recent call last):
__main__.Exception: message
```

Note that *ELLIPSIS* can also be used to ignore the details of the exception message, but such a test may still fail based on whether the module name is present or matches exactly.

Modifié dans la version 3.2 : Maintenant, *IGNORE_EXCEPTION_DETAIL* permet aussi d'ignorer toute information liée au module contenant l'exception qui est en train d'être testée.

doctest.**SKIP**

Lorsque précisé, cesse complètement l'exécution de tous les exemples. Ceci peut être utile dans des contextes où les exemples *doctest* sont à la fois de la documentation et des cas de tests, et qu'un exemple doit être inclus pour des raisons de documentation, mais ne devrait pas être vérifié. Par exemple, la sortie de l'exemple doit être aléatoire ; ou encore, lorsque l'exemple peut dépendre de ressources inatteignables pour l'exécuteur de test.

L'option *SKIP* peut aussi être utilisée temporairement afin de commenter des exemples et d'en empêcher l'exécution.

doctest.**COMPARISON_FLAGS**

Un masque binaire effectuant une composition avec OU de toutes les options de comparaisons ci-dessus.

Le deuxième groupe d'options détermine comment les échecs de tests sont signalés :

doctest.**REPORT_UDIFF**

Lorsque précisé, les défaillances qui font intervenir des sorties attendues et réelles multi-lignes sont affichées dans une *diff* unifiée.

doctest.**REPORT_CDIFF**

Lorsque précisé, les défaillances qui font intervenir des sorties attendues et réelles multi-lignes sont affichées dans une *diff* de contexte.

doctest.**REPORT_NDIFF**

Lorsque précisé, les différences sont obtenues grâce à `difflib.Differ`, en utilisant le même algorithme que le populaire utilitaire `ndiff.py`. Ceci est la seule méthode qui puisse faire la différence à l'intérieur des lignes ainsi que parmi les lignes prises conjointement. Par exemple, si une ligne de sortie attendue contient le chiffre `1` alors que la sortie réelle contient la lettre `l`, une ligne est insérée avec un marqueur caret démarquant les positions de colonnes où il n'y a pas de correspondance.

doctest.**REPORT_ONLY_FIRST_FAILURE**

Lorsque précisé, le premier exemple défaillant de chaque *doctest* est affiché, mais la sortie est supprimée pour tous les autres exemples. Ceci empêche *doctest* de rapporter les exemples adéquats qui échouent du fait d'échecs précédents ; ceci peut aussi cacher des exemples inadéquats qui échouent de façon indépendante au premier échec. Lorsque *REPORT_ONLY_FIRST_FAILURE* est précisé, les exemples restants sont toujours exécutés, et sont toujours comptabilisés dans le nombre total des lignes échouant ; seulement la sortie est omise.

doctest.**FAIL_FAST**

Lorsque précisé, mettre fin à l'exécution après le premier exemple défaillant et ne pas essayer d'exécuter les exemples restants. Ainsi, le nombre d'échecs rapporté sera au plus un (1). Cette option peut être utile durant le débogage, étant donné que les exemples suivant le premier échec ne produiront aucune sortie de débogage.

La ligne de commande de *doctest* accepte l'option `-f` comme un raccourci de `-o FAIL_FAST`.

Ajouté dans la version 3.4.

doctest.**REPORTING_FLAGS**

Un masque binaire effectuant une composition avec le OU de toutes les options de signalement ci-dessus.

Il y a aussi une façon d'enregistrer des nouveaux noms d'option, quoique ceci n'est pas utile sauf dans le cas où vous devez faire une extension pour le code interne de *doctest* par le biais d'une sous-classe :

doctest.**register_optionflag**(*name*)

Crée une nouvelle option avec un nom donné, et renvoie la valeur en nombre entier de la nouvelle option. La fonction *register_optionflag()* peut être utilisée lors de la création de sous-classes à partir de *OutputChecker* ou *DocTestRunner* pour créer de nouvelles options qui sont supportées par vos sous-classes. La fonction *register_optionflag()* devrait toujours être appelée par l'expression suivante :

```
MY_FLAG = register_optionflag('MY_FLAG')
```

Instructions

Les instructions *doctest* peuvent être utilisées afin de modifier les *options* pour un exemple individuel. Les instructions *doctest* sont des commentaires Python spéciaux suivant le code source d'un exemple :

```
directive:             "#" "doctest:" directive_options
directive_options:     directive_option ("," directive_option)*
directive_option:      on_or_off directive_option_name
on_or_off:             "+" | "-"
directive_option_name: "DONT_ACCEPT_BLANKLINE" | "NORMALIZE_WHITESPACE" | ...
```

Les caractères d'espacement ne sont pas permis entre les `+` ou les `-` et le nom de l'option d'instruction. Le nom de l'option d'instruction peut être n'importe lequel des noms d'options expliqués ci-dessus.

Les instructions d'un exemple *doctest* modifient le comportement de *doctest* et ce, seulement pour cet exemple. Utilisez + pour activer le comportement nommé, ou – pour le désactiver.

For example, this test passes :

```
>>> print(list(range(20)))  # doctest: +NORMALIZE_WHITESPACE
[0,   1,  2,  3,  4,  5,  6,  7,  8,  9,
10,  11, 12, 13, 14, 15, 16, 17, 18, 19]
```

Without the directive it would fail, both because the actual output doesn't have two blanks before the single-digit list elements, and because the actual output is on a single line. This test also passes, and also requires a directive to do so :

```
>>> print(list(range(20)))  # doctest: +ELLIPSIS
[0, 1, ..., 18, 19]
```

Multiple directives can be used on a single physical line, separated by commas :

```
>>> print(list(range(20)))  # doctest: +ELLIPSIS, +NORMALIZE_WHITESPACE
[0,    1, ...,   18,    19]
```

If multiple directive comments are used for a single example, then they are combined :

```
>>> print(list(range(20)))  # doctest: +ELLIPSIS
...                         # doctest: +NORMALIZE_WHITESPACE
[0,    1, ...,   18,    19]
```

As the previous example shows, you can add `...` lines to your example containing only directives. This can be useful when an example is too long for a directive to comfortably fit on the same line :

```
>>> print(list(range(5)) + list(range(10, 20)) + list(range(30, 40)))
... # doctest: +ELLIPSIS
[0, ..., 4, 10, ..., 19, 30, ..., 39]
```

Prendre note que puisque toutes les options sont désactivées par défaut, et comme les instructions s'appliquent uniquement aux exemples dans lesquelles elles apparaissent, activer les options (par le biais de + dans une instruction) est habituellement le seul choix ayant du sens. Toutefois, les options peuvent aussi être passées aux fonctions qui exécutent les *doctests*, définissant de nouvelles valeurs par défaut. Dans de tels cas, désactiver une option par l'utilisation de – dans une instruction peut être utile.

Avertissements

Le module `doctest` est rigoureux pour ce qui est d'inclure des correspondances exactes dans la sortie attendue. Si un seul caractère ne correspond pas, le test échoue. Ceci vous surprendra probablement quelques fois, alors que vous apprenez exactement ce que Python garantit et ne garantit pas pour qui est des sorties. Par exemple, lorsqu'on affiche un ensemble (set), Python ne garantit pas que les éléments sont affichés dans un ordre particulier ; ainsi un test tel que

```
>>> foo()
{"spam", "eggs"}
```

est vulnérable ! Une alternative est de faire

```
>>> foo() == {"spam", "eggs"}
True
```

à la place. Une autre façon de faire est

```
>>> d = sorted(foo())
>>> d
['eggs', 'spam']
```

Il y en a d'autres, mais vous saisissez l'idée.

Another bad idea is to print things that embed an object address, like

```
>>> id(1.0)  # certain to fail some of the time
7948648
>>> class C: pass
>>> C()  # the default repr() for instances embeds an address
<C object at 0x00AC18F0>
```

The `ELLIPSIS` directive gives a nice approach for the last example :

```
>>> C()  # doctest: +ELLIPSIS
<C object at 0x...>
```

Les nombres à virgule flottante sont aussi sujets à de petites variations à la sortie, tout dépendamment de la plateforme utilisée, étant donné que Python s'en remet à la bibliothèque de la plateforme C pour la mise-en-forme des *floats*, et les bibliothèques C varient grandement pour ce qui de leur qualité sur ce point.

```
>>> 1./7  # risky
0.14285714285714285
>>> print(1./7) # safer
0.142857142857
>>> print(round(1./7, 6)) # much safer
0.142857
```

Numbers of the form `I/2.**J` are safe across all platforms, and I often contrive doctest examples to produce numbers of that form :

```
>>> 3./4  # utterly safe
0.75
```

Les fractions simples sont aussi plus faciles à comprendre, et cela fait une meilleure documentation.

27.4.4 API de base

Les fonctions `testmod()` et `testfile()` fournissent une interface simple pour *doctest* qui est suffisante pour les cas d'usage les plus élémentaires. Pour une introduction moins formelle à ces deux fonctions, voir les sections *Utilisation simple : vérifier des exemples dans des docstrings* et *Utilisation simple : vérifier des exemples dans un fichier texte*.

`doctest.`**`testfile`**(*filename*, *module_relative=True*, *name=None*, *package=None*, *globs=None*, *verbose=None*, *report=True*, *optionflags=0*, *extraglobs=None*, *raise_on_error=False*, *parser=DocTestParser()*, *encoding=None*)

Tous les arguments sauf *filename* sont optionnels, et doivent être précisés sous forme lettrée.

Teste les exemples dans le fichier nommé *filename*. Renvoie `(failure_count, test_count)`.

L'argument optionnel *module_relative* précise comment le nom de fichier doit être interprété :

— Si *module_relative* prend la valeur `True` (la valeur par défaut), alors *filename* précise un chemin relatif au module qui soit indépendant du système d'exploitation (*OS*). Par défaut, ce chemin est relatif au répertoire du module appelant ; mais si l'argument *package* est précisé, alors il est relatif à ce paquet. Pour garantir l'indépendance quant au système d'exploitation, *filename* doit utiliser des caractères `/` pour séparer chaque segment de chemin, et ne peut pas être un chemin absolu (c'est-à-dire qu'il ne peut pas commencer par `/`).

— Si *module_relative* prend la valeur `False`, alors *filename* précise un chemin en fonction du système d'exploitation. Le chemin peut être absolu ou relatif ; les chemins relatifs sont résolus en rapport au répertoire actif.

L'option *name* désigne le nom du test ; par défaut, ou si `None` est passé en argument, `os.path.basename(filename)` est utilisé.

L'option *package* est un paquet Python ou le nom d'un paquet Python dont le répertoire doit être utilisé comme le répertoire principal pour un nom de fichier lié à un module. Si aucun paquet n'est spécifié, le répertoire du

module appelé à l'exécution est utilisé comme le répertoire principal pour les noms de fichiers liés au module. C'est une erreur que de spécifier *package* si *module_relative* a `False` comme valeur.

L'option *globs* spécifie un *dict* à utiliser comme *globals* lorsque des exemples sont exécutés. Une copie superficielle de ce *dict* est créée pour le *doctest* ; ainsi, ces exemples commencent avec un état vide. Par défaut, ou si `None` est passé en argument, un nouveau *dict* vide est utilisé.

L'option *extraglobs* spécifie un *dict* intégré dans les variables globales utilisées pour exécuter l'exemple. Ceci fonctionne comme `dict.update()` : si *globs* et *extraglobs* ont une clé commune, la valeur associée à *extraglobs* apparaît dans le *dict* combiné. Par défaut, ou si `None` est passé en argument, aucune variable globale supplémentaire est utilisée. Ceci est une fonctionnalité avancée qui permet la configuration des *doctests*. Par exemple, un *doctest* peut être rédigé pour une classe de base, en utilisant un nom générique pour la classe, puis réutilisé afin de tester un nombre indéfini de sous-classes en passant un *dict extraglobs* reliant le nom générique à la sous-classe qui doit être testée.

L'option *verbose* affiche une grande quantité d'information si elle est vraie, et affiche uniquement les défaillances si elle est fausse ; par défaut, ou si `None`, celle-ci est vraie si et seulement si `-v` est présent dans `sys.argv`.

L'option *report* affiche un résumé à la fin lorsque vraie ; sinon, rien n'est affiché à la fin. En mode *verbose*, le résumé est détaillé, sinon le résumé est très bref (en fait, vide si tous les tests ont réussi).

L'option *optionflags* (dont la valeur par défaut est de zéro) calcule la valeur bitwise OR des options de ligne de commande. Voir la section *Options de ligne de commande*.

L'option *raise_on_error* est fausse par défaut. Si elle est vraie, une exception est levée à la première défaillance ou à la première exception qui ne soit prévue dans l'exemple. Ceci permet aux défaillances d'être analysées lors d'un post-mortem. Le comportement par défaut est de poursuivre l'exécution des exemples.

L'option *parser* définit une classe ou une sous-classe `DocTestParser` qui doit être utilisée pour extraire les tests des fichiers. Par défaut, on utilise un analyseur normal (c'est-à-dire, `DocTestParser()`).

L'option *encoding* définit un encodage à utiliser pour convertir le fichier en format *unicode*.

`doctest.`**`testmod`**(*m=None*, *name=None*, *globs=None*, *verbose=None*, *report=True*, *optionflags=0*, *extraglobs=None*, *raise_on_error=False*, *exclude_empty=False*)

Toutes les options sont facultatives, et toutes sauf *m* doivent être définies en format lettré.

Ceci teste les exemples en *docstrings* dans les fonctions et les classes accessibles depuis le module *m* (ou depuis le module `__main__` si *m* n'a pas été défini ou est `None`), en commençant par `m.__doc__`.

Also test examples reachable from dict `m.__test__`, if it exists. `m.__test__` maps names (strings) to functions, classes and strings ; function and class docstrings are searched for examples ; strings are searched directly, as if they were docstrings.

Seulement les *docstrings* attribuées à des objets appartenant au module *m* sont fouillées.

Renvoie `(failure_count, test_count)`.

L'option *name* donne le nom du module ; par défaut, ou si `None`, `m.__name__` est utilisé.

Optional argument *exclude_empty* defaults to false. If true, objects for which no doctests are found are excluded from consideration. The default is a backward compatibility hack, so that code still using `doctest.master.summarize` in conjunction with `testmod()` continues to get output for objects with no tests. The *exclude_empty* argument to the newer `DocTestFinder` constructor defaults to true.

Les options *extraglobs*, *verbose*, *report*, *optionflags*, *raise_on_error*, et *globs* sont les mêmes que pour la fonction `testfile()` ci-dessus, sauf pour *globs*, qui est `m.__dict__` par défaut.

`doctest.`**`run_docstring_examples`**(*f*, *globs*, *verbose=False*, *name='NoName'*, *compileflags=None*, *optionflags=0*)

Les exemples de test associés à l'objet *f* ; par exemple, *f* peut être une chaîne de caractères, un module, une fonction, ou un objet de classe.

Une copie superficielle de l'argument-dictionnaire *globs* est utilisée pour le contexte d'exécution.

L'option *name* est utilisée pour les messages d'échec, et prend `"NoName"` comme valeur par défaut.

Si l'option *verbose* est vraie, les sorties sont générées même s'il n'y a aucun échec. Par défaut, la sortie est

générée seulement si un exemple échoue.

L'option *compileflags* donne l'ensemble des options qui doit être utilisée par le compilateur Python lorsqu'il exécute les exemples. Par défaut, ou si `None`, les options sont inférées à partir de l'ensemble des fonctionnalités futures trouvées dans *globs*.

L'option *optionflags* fonctionne similairement à la fonction `testfile()` ci-dessus.

27.4.5 API de tests unitaires

As your collection of doctest'ed modules grows, you'll want a way to run all their doctests systematically. `doctest` provides two functions that can be used to create `unittest` test suites from modules and text files containing doctests. To integrate with `unittest` test discovery, include a *load_tests* function in your test module :

```
import unittest
import doctest
import my_module_with_doctests

def load_tests(loader, tests, ignore):
    tests.addTests(doctest.DocTestSuite(my_module_with_doctests))
    return tests
```

Il y a deux fonctions principales pour créer des instances de la classe `unittest.TestSuite` à partir de fichiers textes et de modules ayant des *doctests* :

doctest.**DocFileSuite**(**paths*, *module_relative=True*, *package=None*, *setUp=None*, *tearDown=None*, *globs=None*, *optionflags=0*, *parser=DocTestParser()*, *encoding=None*)

Convertit des tests *doctest* à partir d'un ou plusieurs fichiers vers une classe `unittest.TestSuite`.

The returned `unittest.TestSuite` is to be run by the unittest framework and runs the interactive examples in each file. If an example in any file fails, then the synthesized unit test fails, and a `failureException` exception is raised showing the name of the file containing the test and a (sometimes approximate) line number. If all the examples in a file are skipped, then the synthesized unit test is also marked as skipped.

Passe un ou plusieurs chemins (sous forme de chaînes de caractères) à des fichiers textes afin d'être vérifiés.

Les options peuvent être fournies comme des options lettrées :

L'option *module_relative* précise comment les noms de fichiers dans *paths* doivent être interprétés :

— Si *module_relative* a `True` comme valeur (valeur par défaut), alors chaque nom de fichier dans *paths* précise un chemin relatif au module qui soit indépendant du système d'exploitation. Par défaut, ce chemin est relatif au répertoire du module appelant ; mais si l'option *package* est précisée, alors il est relatif à ce paquet. Afin de garantir l'indépendance face au système d'exploitation, chaque nom de fichier doit utiliser des caractères `/` afin de séparer les segments de chemin, et ne peut pas être un chemin absolu (c'est-à-dire, il ne peut pas commencer par `/`).

— Si *module_relative* prend la valeur `False`, alors *filename* précise un chemin en fonction du système d'exploitation. Le chemin peut être absolu ou relatif ; les chemins relatifs sont résolus en rapport au répertoire actif.

L'option *package* est un paquet Python ou le nom d'un paquet Python dont le répertoire dont être utilisé comme le répertoire principal pour un nom de fichier dans *paths* qui soit lié à un module. Si aucun paquet n'est spécifié, le répertoire du module appelé à l'exécution est utilisé comme le répertoire principal pour les noms de fichiers liés au module. C'est une erreur que de spécifier *package* si *module_relative* a `False` comme valeur.

L'option *setUp* précise une fonction de mise-en-place pour la suite de tests. Ceci est appelé avant l'exécution des tests dans chaque fichier. La fonction *setUp* est passée à un objet `DocTest`. La fonction *setUp* peut accéder aux valeurs globales du test par le biais de l'attribut *globs* du test passé.

L'option *tearDown* précise une fonction de démolition pour la suite de tests. Celle-ci est appelée après avoir exécuté les tests dans chaque fichier. La fonction *tearDown* est passée à un objet `DocTest`. La fonction *setUp* peut accéder aux valeurs globales du test par l'attribut *globs* du test passé.

L'option *globs* est un dictionnaire contenant les variables globales initiales pour les tests. Une nouvelle copie de ce dictionnaire est créée pour chaque test. Par défaut, *globs* est un nouveau dictionnaire vide.

Les options *optionflags* précisent les options par défaut de *doctest* pour les tests, créées en composant par un OU les différentes options individuelles. Voir la section *Options de ligne de commande*. Voir la fonction `set_unittest_reportflags()` ci-dessous pour une meilleure façon de préciser des options de rapport.

L'option *parser* définit une classe ou une sous-classe `DocTestParser` qui doit être utilisée pour extraire les tests des fichiers. Par défaut, on utilise un analyseur normal (c'est-à-dire, `DocTestParser()`).

L'option *encoding* définit un encodage à utiliser pour convertir le fichier en format *unicode*.

La valeur globale `__file__` est ajoutée aux valeurs globales fournies par les *doctests*, ceux-ci étant téléchargés d'un fichier texte utilisant la fonction `DocFileSuite()`.

`doctest.`**`DocTestSuite`**(*module=None*, *globs=None*, *extraglobs=None*, *test_finder=None*, *setUp=None*, *tearDown=None*, *optionflags=0*, *checker=None*)

Convertit les tests *doctest* pour un module donné à une classe `unittest.TestSuite`.

The returned `unittest.TestSuite` is to be run by the unittest framework and runs each doctest in the module. If any of the doctests fail, then the synthesized unit test fails, and a `failureException` exception is raised showing the name of the file containing the test and a (sometimes approximate) line number. If all the examples in a docstring are skipped, then the synthesized unit test is also marked as skipped.

L'option *module* fournit le module qui sera testé. Il peut prendre la forme d'un objet-module ou celle du nom d'un module (possiblement *dotted*). Si non-précisée, le module appelant cette fonction est utilisé.

L'option *globs* est un dictionnaire contenant les variables globales initiales pour les tests. Une nouvelle copie de ce dictionnaire est créée pour chaque test. Par défaut, *globs* est un nouveau dictionnaire vide.

L'option *extraglobs* précise un ensemble supplémentaire de variables globales, à fusionner avec *globs*. Par défaut, aucune variable globale supplémentaire est utilisée.

L'option *test_finder* est l'instance de `DocTestFinder` (ou un remplacement *drop-in*) qui est utilisée pour extraire les *doctests* à partir du module.

Les options *setUp*, *tearDown* et *optionflags* sont les mêmes que pour la fonction `DocFileSuite()` ci-dessus.

Cette fonction utilise la même technique de recherche que `testmod()`.

Modifié dans la version 3.5 : La fonction `DocTestSuite()` renvoie une instance vide de la classe `unittest.TestSuite` si *module* ne contient aucune *docstring*, et ce, au lieu de lever l'exception `ValueError`.

exception `doctest.`**`failureException`**

When doctests which have been converted to unit tests by `DocFileSuite()` or `DocTestSuite()` fail, this exception is raised showing the name of the file containing the test and a (sometimes approximate) line number.

Under the covers, `DocTestSuite()` creates a `unittest.TestSuite` out of `doctest.DocTestCase` instances, and `DocTestCase` is a subclass of `unittest.TestCase`. `DocTestCase` isn't documented here (it's an internal detail), but studying its code can answer questions about the exact details of `unittest` integration.

Similarly, `DocFileSuite()` creates a `unittest.TestSuite` out of `doctest.DocFileCase` instances, and `DocFileCase` is a subclass of `DocTestCase`.

So both ways of creating a `unittest.TestSuite` run instances of `DocTestCase`. This is important for a subtle reason : when you run `doctest` functions yourself, you can control the `doctest` options in use directly, by passing option flags to `doctest` functions. However, if you're writing a `unittest` framework, `unittest` ultimately controls when and how tests get run. The framework author typically wants to control `doctest` reporting options (perhaps, e.g., specified by command line options), but there's no way to pass options through `unittest` to `doctest` test runners.

Pour cette raison, `doctest` implémente le concept d'options de rapport de `doctest` qui soit spécifique à `unittest`, par le biais de cette fonction :

`doctest.`**`set_unittest_reportflags`**(*flags*)

Voir les options de ligne de commandes de `doctest` pour l'utilisation.

L'argument *flags* prend la composition OU bit à bit des arguments. Voir la section *Options de ligne de commande*. Ce sont uniquement les "options de rapport" qui peuvent être utilisées.

This is a module-global setting, and affects all future doctests run by module `unittest` : the `runTest()` method of `DocTestCase` looks at the option flags specified for the test case when the `DocTestCase` instance was constructed. If no reporting flags were specified (which is the typical and expected case), `doctest`'s `unittest` reporting flags are bitwise ORed into the option flags, and the option flags so augmented are passed to the `DocTestRunner` instance created to run the doctest. If any reporting flags were specified when the `DocTestCase` instance was constructed, `doctest`'s `unittest` reporting flags are ignored.

La valeur prise par les options de rapport de `unittest` et ce, avant que la fonction n'ait été appelée, est renvoyée par la fonction.

27.4.6 API avancé

The basic API is a simple wrapper that's intended to make doctest easy to use. It is fairly flexible, and should meet most users' needs ; however, if you require more fine-grained control over testing, or wish to extend doctest's capabilities, then you should use the advanced API.

The advanced API revolves around two container classes, which are used to store the interactive examples extracted from doctest cases :

- `Example` : Un unique *statement* Python, pris conjointement avec sa sortie attendue.
- `DocTest` : Une collection de `Example`, habituellement extraits d'une seule *docstring* ou fichier texte.

Additional processing classes are defined to find, parse, and run, and check doctest examples :

- `DocTestFinder` : Finds all docstrings in a given module, and uses a `DocTestParser` to create a `DocTest` from every docstring that contains interactive examples.
- `DocTestParser` : Creates a `DocTest` object from a string (such as an object's docstring).
- `DocTestRunner` : Executes the examples in a `DocTest`, and uses an `OutputChecker` to verify their output.
- `OutputChecker` : Compares the actual output from a doctest example with the expected output, and decides whether they match.

The relationships among these processing classes are summarized in the following diagram :

```
                             list of:
+------+                    +---------+
|module| --DocTestFinder-> | DocTest | --DocTestRunner-> results
+------+    |        ^     +---------+     |       ^     (printed)
            |        |     | Example |     |       |
            v        |     |   ...   |     v       |
           DocTestParser   | Example |   OutputChecker
                            +---------+
```

Objets *doctest*

class doctest.**DocTest**(*examples*, *globs*, *name*, *filename*, *lineno*, *docstring*)

A collection of doctest examples that should be run in a single namespace. The constructor arguments are used to initialize the attributes of the same names.

`DocTest` définit les attributs suivants. Ils sont initialisés par le constructeur, et ne doivent pas être modifiés directement.

examples

A list of `Example` objects encoding the individual interactive Python examples that should be run by this test.

globs

The namespace (aka globals) that the examples should be run in. This is a dictionary mapping names to values. Any changes to the namespace made by the examples (such as binding new variables) will be reflected in `globs` after the test is run.

name

A string name identifying the `DocTest`. Typically, this is the name of the object or file that the test was extracted from.

filename

The name of the file that this `DocTest` was extracted from ; or `None` if the filename is unknown, or if the `DocTest` was not extracted from a file.

lineno

The line number within `filename` where this `DocTest` begins, or `None` if the line number is unavailable. This line number is zero-based with respect to the beginning of the file.

docstring

The string that the test was extracted from, or `None` if the string is unavailable, or if the test was not extracted from a string.

Exemples d'objets

class doctest.**Example**(*source*, *want*, *exc_msg=None*, *lineno=0*, *indent=0*, *options=None*)

A single interactive example, consisting of a Python statement and its expected output. The constructor arguments are used to initialize the attributes of the same names.

`Example` defines the following attributes. They are initialized by the constructor, and should not be modified directly.

source

A string containing the example's source code. This source code consists of a single Python statement, and always ends with a newline ; the constructor adds a newline when necessary.

want

The expected output from running the example's source code (either from stdout, or a traceback in case of exception). `want` ends with a newline unless no output is expected, in which case it's an empty string. The constructor adds a newline when necessary.

exc_msg

The exception message generated by the example, if the example is expected to generate an exception ; or `None` if it is not expected to generate an exception. This exception message is compared against the return value of `traceback.format_exception_only()`. `exc_msg` ends with a newline unless it's `None`. The constructor adds a newline if needed.

lineno

The line number within the string containing this example where the example begins. This line number is zero-based with respect to the beginning of the containing string.

indent

The example's indentation in the containing string, i.e., the number of space characters that precede the example's first prompt.

options

A dictionary mapping from option flags to `True` or `False`, which is used to override default options for this example. Any option flags not contained in this dictionary are left at their default value (as specified by the `DocTestRunner`'s *optionflags*). By default, no options are set.

Objets *DocTestFinder*

class doctest.**DocTestFinder**(*verbose=False*, *parser=DocTestParser()*, *recurse=True*, *exclude_empty=True*)

A processing class used to extract the `DocTest`s that are relevant to a given object, from its docstring and the docstrings of its contained objects. `DocTest`s can be extracted from modules, classes, functions, methods, staticmethods, classmethods, and properties.

The optional argument *verbose* can be used to display the objects searched by the finder. It defaults to `False` (no output).

The optional argument *parser* specifies the `DocTestParser` object (or a drop-in replacement) that is used to extract doctests from docstrings.

If the optional argument *recurse* is false, then `DocTestFinder.find()` will only examine the given object, and not any contained objects.

If the optional argument *exclude_empty* is false, then `DocTestFinder.find()` will include tests for objects with empty docstrings.

La classe `DocTestFinder` définit la méthode suivante :

find(*obj[, name][, module][, globs][, extraglobs]*)

Return a list of the `DocTest`s that are defined by *obj*'s docstring, or by any of its contained objects' docstrings.

The optional argument *name* specifies the object's name ; this name will be used to construct names for the returned `DocTest`s. If *name* is not specified, then `obj.__name__` is used.

The optional parameter *module* is the module that contains the given object. If the module is not specified or is `None`, then the test finder will attempt to automatically determine the correct module. The object's module is used :

— As a default namespace, if *globs* is not specified.
— To prevent the DocTestFinder from extracting DocTests from objects that are imported from other modules. (Contained objects with modules other than *module* are ignored.)
— Afin de trouver le nom du fichier contenant l'objet.
— To help find the line number of the object within its file.

If *module* is `False`, no attempt to find the module will be made. This is obscure, of use mostly in testing doctest itself : if *module* is `False`, or is `None` but cannot be found automatically, then all objects are considered to belong to the (non-existent) module, so all contained objects will (recursively) be searched for doctests.

The globals for each `DocTest` is formed by combining *globs* and *extraglobs* (bindings in *extraglobs* override bindings in *globs*). A new shallow copy of the globals dictionary is created for each `DocTest`. If *globs* is not specified, then it defaults to the module's *__dict__*, if specified, or `{}` otherwise. If *extraglobs* is not specified, then it defaults to `{}`.

Objets *DocTestParser*

class doctest.**DocTestParser**

A processing class used to extract interactive examples from a string, and use them to create a `DocTest` object.

La classe `DocTestFinder` définit les méthodes suivantes :

get_doctest(*string*, *globs*, *name*, *filename*, *lineno*)

Extrait tous les exemples de *doctests* à partir de la chaîne de caractère donnée, et les réunit dans un objet `DocTest`.

Notez que *globs*, *name*, *filname* et *lineno* sont des attributs pour le nouvel objet `DocTest`. Voir la documentation pour `DocTest` pour plus d'information.

get_examples(*string*, *name='<string>'*)

Extract all doctest examples from the given string, and return them as a list of `Example` objects. Line numbers are 0-based. The optional argument *name* is a name identifying this string, and is only used for error messages.

parse(*string*, *name='<string>'*)

Divide the given string into examples and intervening text, and return them as a list of alternating `Example`s and strings. Line numbers for the `Example`s are 0-based. The optional argument *name* is a name identifying this string, and is only used for error messages.

Objets *TestResults*

class doctest.**TestResults**(*failed*, *attempted*)

failed

Nombre de tests échoués.

attempted

Nombre de tests tentés.

skipped

Nombre de tests sautés.

Ajouté dans la version 3.13.

Objets *DocTestRunner*

class doctest.**DocTestRunner**(*checker=None*, *verbose=None*, *optionflags=0*)

A processing class used to execute and verify the interactive examples in a `DocTest`.

The comparison between expected outputs and actual outputs is done by an `OutputChecker`. This comparison may be customized with a number of option flags ; see section *Options de ligne de commande* for more information. If the option flags are insufficient, then the comparison may also be customized by passing a subclass of `OutputChecker` to the constructor.

The test runner's display output can be controlled in two ways. First, an output function can be passed to `run()` ; this function will be called with strings that should be displayed. It defaults to `sys.stdout.write`. If capturing the output is not sufficient, then the display output can be also customized by subclassing DocTestRunner, and overriding the methods `report_start()`, `report_success()`, `report_unexpected_exception()`, and `report_failure()`.

The optional keyword argument *checker* specifies the `OutputChecker` object (or drop-in replacement) that should be used to compare the expected outputs to the actual outputs of doctest examples.

The optional keyword argument *verbose* controls the `DocTestRunner`'s verbosity. If *verbose* is `True`, then information is printed about each example, as it is run. If *verbose* is `False`, then only failures are printed. If *verbose* is unspecified, or `None`, then verbose output is used iff the command-line switch `-v` is used.

The optional keyword argument *optionflags* can be used to control how the test runner compares expected output to actual output, and how it displays failures. For more information, see section *Options de ligne de commande*.

The test runner accumulates statistics. The aggregated number of attempted, failed and skipped examples is also available via the `tries`, `failures` and `skips` attributes. The `run()` and `summarize()` methods return a `TestResults` instance.

La classe `DocTestRunner` définit les méthodes suivantes :

report_start(*out*, *test*, *example*)

Report that the test runner is about to process the given example. This method is provided to allow subclasses of `DocTestRunner` to customize their output ; it should not be called directly.

example is the example about to be processed. *test* is the test *containing example*. *out* is the output function that was passed to `DocTestRunner.run()`.

report_success(*out*, *test*, *example*, *got*)

Report that the given example ran successfully. This method is provided to allow subclasses of `DocTestRunner` to customize their output ; it should not be called directly.

example is the example about to be processed. *got* is the actual output from the example. *test* is the test containing *example*. *out* is the output function that was passed to `DocTestRunner.run()`.

report_failure(*out*, *test*, *example*, *got*)

Report that the given example failed. This method is provided to allow subclasses of `DocTestRunner` to customize their output ; it should not be called directly.

example is the example about to be processed. *got* is the actual output from the example. *test* is the test containing *example*. *out* is the output function that was passed to `DocTestRunner.run()`.

report_unexpected_exception(*out*, *test*, *example*, *exc_info*)

Report that the given example raised an unexpected exception. This method is provided to allow subclasses of `DocTestRunner` to customize their output ; it should not be called directly.

example is the example about to be processed. *exc_info* is a tuple containing information about the unexpected exception (as returned by `sys.exc_info()`). *test* is the test containing *example*. *out* is the output function that was passed to `DocTestRunner.run()`.

run(*test*, *compileflags=None*, *out=None*, *clear_globs=True*)

Run the examples in *test* (a `DocTest` object), and display the results using the writer function *out*. Return a `TestResults` instance.

The examples are run in the namespace `test.globs`. If *clear_globs* is true (the default), then this namespace will be cleared after the test runs, to help with garbage collection. If you would like to examine the namespace after the test completes, then use *clear_globs=False*.

compileflags gives the set of flags that should be used by the Python compiler when running the examples. If not specified, then it will default to the set of future-import flags that apply to *globs*.

The output of each example is checked using the `DocTestRunner`'s output checker, and the results are formatted by the `DocTestRunner.report_*()` methods.

summarize(*verbose=None*)

Print a summary of all the test cases that have been run by this DocTestRunner, and return a `TestResults` instance.

The optional *verbose* argument controls how detailed the summary is. If the verbosity is not specified, then the `DocTestRunner`'s verbosity is used.

`DocTestParser` définit les attributs suivants :

tries

Number of attempted examples.

failures

Number of failed examples.

skips

Number of skipped examples.

Ajouté dans la version 3.13.

Objets *OutputChecker*

class doctest.**OutputChecker**

A class used to check the whether the actual output from a doctest example matches the expected output. `OutputChecker` defines two methods : `check_output()`, which compares a given pair of outputs, and returns `True` if they match ; and `output_difference()`, which returns a string describing the differences between two outputs.

La classe `OutputChecker` définit les méthodes suivantes :

check_output(*want*, *got*, *optionflags*)

Return `True` iff the actual output from an example (*got*) matches the expected output (*want*). These strings are always considered to match if they are identical ; but depending on what option flags the test runner is using, several non-exact match types are also possible. See section *Options de ligne de commande* for more information about option flags.

output_difference(*example*, *got*, *optionflags*)

Return a string describing the differences between the expected output for a given example (*example*) and the actual output (*got*). *optionflags* is the set of option flags used to compare *want* and *got*.

27.4.7 Débogage

Doctest fournit plusieurs mécanismes pour déboguer des exemples *doctest* :

— Plusieurs fonctions convertissent les *doctests* en programmes Python exécutables, qui peuvent être exécutés grâce au débogueur Python, `pdb`.

— The `DebugRunner` class is a subclass of `DocTestRunner` that raises an exception for the first failing example, containing information about that example. This information can be used to perform post-mortem debugging on the example.
— The `unittest` cases generated by `DocTestSuite()` support the `debug()` method defined by `unittest.TestCase`.
— You can add a call to `pdb.set_trace()` in a doctest example, and you'll drop into the Python debugger when that line is executed. Then you can inspect current values of variables, and so on. For example, suppose `a.py` contains just this module docstring :

```
"""
>>> def f(x):
...     g(x*2)
>>> def g(x):
...     print(x+3)
...     import pdb; pdb.set_trace()
>>> f(3)
9
"""
```

Alors une séance interactive de Python peut ressembler à ceci :

```
>>> import a, doctest
>>> doctest.testmod(a)
--Return--
> <doctest a[1]>(3)g()->None
-> import pdb; pdb.set_trace()
(Pdb) list
  1     def g(x):
  2         print(x+3)
  3  ->     import pdb; pdb.set_trace()
[EOF]
(Pdb) p x
6
(Pdb) step
--Return--
> <doctest a[0]>(2)f()->None
-> g(x*2)
(Pdb) list
  1     def f(x):
  2  ->     g(x*2)
[EOF]
(Pdb) p x
3
(Pdb) step
--Return--
> <doctest a[2]>(1)?()->None
-> f(3)
(Pdb) cont
(0, 3)
>>>
```

Functions that convert doctests to Python code, and possibly run the synthesized code under the debugger :

`doctest.`**`script_from_examples`**(*s*)

Convertit du texte contenant des exemples en un script.

Argument *s* is a string containing doctest examples. The string is converted to a Python script, where doctest examples in *s* are converted to regular code, and everything else is converted to Python comments. The generated script is returned as a string. For example,

```
import doctest
print(doctest.script_from_examples(r"""
    Set x and y to 1 and 2.
    >>> x, y = 1, 2

    Print their sum:
    >>> print(x+y)
    3
"""))
```

affiche :

```
# Set x and y to 1 and 2.
x, y = 1, 2
#
# Print their sum:
print(x+y)
# Expected:
## 3
```

Cette fonction est utilisée à l'interne par d'autres fonctions (voir ci-bas), mais peut aussi être utile lorsque l'on souhaite transformer une séance interactive de Python en script Python.

`doctest.`**`testsource`**(*module*, *name*)

Convertit en script l'objet *doctest*.

Argument *module* is a module object, or dotted name of a module, containing the object whose doctests are of interest. Argument *name* is the name (within the module) of the object with the doctests of interest. The result is a string, containing the object's docstring converted to a Python script, as described for `script_from_examples()` above. For example, if module `a.py` contains a top-level function `f()`, then

```
import a, doctest
print(doctest.testsource(a, "a.f"))
```

prints a script version of function `f()`'s docstring, with doctests converted to code, and the rest placed in comments.

`doctest.`**`debug`**(*module*, *name*, *pm=False*)

Débogue les *doctests* pour un objet.

The *module* and *name* arguments are the same as for function `testsource()` above. The synthesized Python script for the named object's docstring is written to a temporary file, and then that file is run under the control of the Python debugger, `pdb`.

Une copie superficielle de `module.__dict__` est utilisée à la fois pour les contextes d'exécution locaux et globaux.

Optional argument *pm* controls whether post-mortem debugging is used. If *pm* has a true value, the script file is run directly, and the debugger gets involved only if the script terminates via raising an unhandled exception. If it does, then post-mortem debugging is invoked, via `pdb.post_mortem()`, passing the traceback object from the unhandled exception. If *pm* is not specified, or is false, the script is run under the debugger from the start, via passing an appropriate `exec()` call to `pdb.run()`.

`doctest.`**`debug_src`**(*src*, *pm=False*, *globs=None*)

Débogue les *doctests* dans une chaîne de caractères.

Ceci est similaire à la fonction `debug()` décrite ci-haut, mis-à-part qu'une chaîne de caractères contenant des exemples *doctest* est définie directement, par l'option *src*.

L'option *pm* a la même définition que dans la fonction `debug()` ci-haut.

L'option *globs* définit un dictionnaire à utiliser comme contexte d'exécution global et local. Si elle n'est pas définie, ou si `None`, un dictionnaire vide est utilisé. Si définie, une copie superficielle du dictionnaire est utilisée.

The `DebugRunner` class, and the special exceptions it may raise, are of most interest to testing framework authors, and will only be sketched here. See the source code, and especially `DebugRunner`'s docstring (which is a doctest !) for more details :

class doctest.**DebugRunner**(*checker=None*, *verbose=None*, *optionflags=0*)

A subclass of `DocTestRunner` that raises an exception as soon as a failure is encountered. If an unexpected exception occurs, an `UnexpectedException` exception is raised, containing the test, the example, and the original exception. If the output doesn't match, then a `DocTestFailure` exception is raised, containing the test, the example, and the actual output.

Pour de l'information sur les paramètres et méthodes du constructeur, voir la documentation pour la classe `DocTestrunner` dans la section *API avancé*.

Il y a deux exceptions qui peuvent être levées par des instances `DebugRunner` :

exception doctest.**DocTestFailure**(*test*, *example*, *got*)

Une exception levée par `DocTestRunner` pour signaler que la sortie obtenue suite à un exemple *doctest* ne correspond pas à la sortie attendue. Les arguments du constructeur sont utilisés pour initialiser les attributs des mêmes noms.

`DocTestFailure` définit les attributs suivants :

DocTestFailure.**test**

L'objet issu de la classe `DocTest` qui était en cours d'exécution lorsque l'exemple a échoué.

DocTestFailure.**example**

L'exemple `Example` qui a échoué.

DocTestFailure.**got**

La sortie obtenue par l'exécution de l'exemple.

exception doctest.**UnexpectedException**(*test*, *example*, *exc_info*)

Une exception levée par `DocTestRunner` afin de signaler qu'un exemple *doctest* a levé une exception inattendue. Les arguments du constructeur sont utilisés pour initialiser les attributs des mêmes noms.

`UnexpectedException` définit les attributs suivants :

UnexpectedException.**test**

L'objet issu de la classe `DocTest` qui était en cours d'exécution lorsque l'exemple a échoué.

UnexpectedException.**example**

L'exemple `Example` qui a échoué.

UnexpectedException.**exc_info**

Un n-uplet contenant l'information au sujet de l'exception inattendue, telle que retourné par `sys.exc_info()`.

27.4.8 Éditorial

Comme mentionné dans l'introduction, `doctest` a présentement trois usages principaux :

1. Vérifier les exemples dans les *docstrings*.
2. Test de régression.
3. De la documentation exécutable / des tests littéraires.

These uses have different requirements, and it is important to distinguish them. In particular, filling your docstrings with obscure test cases makes for bad documentation.

When writing a docstring, choose docstring examples with care. There's an art to this that needs to be learned---it may not be natural at first. Examples should add genuine value to the documentation. A good example can often be worth many words. If done with care, the examples will be invaluable for your users, and will pay back the time it takes to collect them many times over as the years go by and things change. I'm still amazed at how often one of my `doctest` examples stops working after a "harmless" change.

Doctest also makes an excellent tool for regression testing, especially if you don't skimp on explanatory text. By interleaving prose and examples, it becomes much easier to keep track of what's actually being tested, and why.

When a test fails, good prose can make it much easier to figure out what the problem is, and how it should be fixed. It's true that you could write extensive comments in code-based testing, but few programmers do. Many have found that using doctest approaches instead leads to much clearer tests. Perhaps this is simply because doctest makes writing prose a little easier than writing code, while writing comments in code is a little harder. I think it goes deeper than just that : the natural attitude when writing a doctest-based test is that you want to explain the fine points of your software, and illustrate them with examples. This in turn naturally leads to test files that start with the simplest features, and logically progress to complications and edge cases. A coherent narrative is the result, instead of a collection of isolated functions that test isolated bits of functionality seemingly at random. It's a different attitude, and produces different results, blurring the distinction between testing and explaining.

Regression testing is best confined to dedicated objects or files. There are several options for organizing tests :

— Write text files containing test cases as interactive examples, and test the files using `testfile()` or `DocFileSuite()`. This is recommended, although is easiest to do for new projects, designed from the start to use doctest.

— Define functions named `_regrtest_topic` that consist of single docstrings, containing test cases for the named topics. These functions can be included in the same file as the module, or separated out into a separate test file.

— Define a `__test__` dictionary mapping from regression test topics to docstrings containing test cases.

Lorsque vous placez vos tests dans un module, le module lui-même peut être l'exécuteur de tests. Lorsqu'un test échoue, vous pouvez signifier à votre exécuteur de tests de rouler une seconde fois uniquement les tests qui échouent et ce, tant que vous travaillez sur le problème. Voici un exemple minimal d'un test exécuteur de tests :

```
if __name__ == '__main__':
    import doctest
    flags = doctest.REPORT_NDIFF|doctest.FAIL_FAST
    if len(sys.argv) > 1:
        name = sys.argv[1]
        if name in globals():
            obj = globals()[name]
        else:
            obj = __test__[name]
        doctest.run_docstring_examples(obj, globals(), name=name,
                                       optionflags=flags)
    else:
        fail, total = doctest.testmod(optionflags=flags)
        print(f"{fail} failures out of {total} tests")
```

Notes

27.5 `unittest` --- Unit testing framework

Code source : Lib/unittest/__init__.py

(Si vous êtes déjà familier des concepts de base concernant les tests, vous pouvez souhaiter passer à *la liste des méthodes*.)

Le cadre applicatif de tests unitaires `unittest` était au départ inspiré par *JUnit* et ressemble aux principaux *frameworks* de tests unitaires des autres langages. Il gère l'automatisation des tests, le partage de code pour la mise en place et la finalisation des tests, l'agrégation de tests en collections, et l'indépendance des tests par rapport au *framework* utilisé.

Pour y parvenir, `unittest` gère quelques concepts importants avec une approche orientée objet :

aménagement de test (*fixture*)
: Un *aménagement de test* (*fixture* en anglais) désigne la préparation nécessaire au déroulement d'un ou plusieurs tests, et toutes les actions de nettoyage associées. Cela peut concerner, par exemple, la création de bases de données temporaires ou mandataires, de répertoires, ou le démarrage d'un processus serveur.

scénario de test
Un *scénario de test* est l'élément de base des tests. Il attend une réponse spécifique pour un ensemble particulier d'entrées. `unittest` fournit une classe de base, `TestCase`, qui peut être utilisée pour créer de nouveaux scénarios de test.

suite de tests
Une *suite de tests* est une collection de scénarios de test, de suites de tests ou les deux. Cela sert à regrouper les tests qui devraient être exécutés ensemble.

lanceur de tests
Un *lanceur de tests* est un composant qui orchestre l'exécution des tests et fournit le résultat pour l'utilisateur. Le lanceur peut utiliser une interface graphique, une interface textuelle, ou renvoie une valeur spéciale pour indiquer les résultats de l'exécution des tests.

Voir aussi

Module `doctest`
Un autre module de test adoptant une approche très différente.

Simple Smalltalk Testing : With Patterns
Le papier originel de Kent Beck sur les *frameworks* de test utilisant le modèle sur lequel s'appuie `unittest`.

pytest
Des cadres applicatifs tiers de tests unitaires avec une syntaxe allégée pour l'écriture des tests. Par exemple, `assert func(10) == 42`.

The Python Testing Tools Taxonomy
Une liste étendue des outils de test pour Python comprenant des *frameworks* de tests fonctionnels et des bibliothèques d'objets simulés (*mocks*).

Testing in Python Mailing List
Un groupe de discussion dédié aux tests, et outils de test, en Python.

The script `Tools/unittestgui/unittestgui.py` in the Python source distribution is a GUI tool for test discovery and execution. This is intended largely for ease of use for those new to unit testing. For production environments it is recommended that tests be driven by a continuous integration system such as Buildbot, Jenkins, GitHub Actions, or AppVeyor.

27.5.1 Exemple basique

Le module `unittest` fournit un riche ensemble d'outils pour construire et lancer des tests. Cette section montre qu'une petite partie des outils suffit pour satisfaire les besoins de la plupart des utilisateurs.

Voici un court script pour tester trois méthodes de *string* :

```
import unittest

class TestStringMethods(unittest.TestCase):

    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])
        # check that s.split fails when the separator is not a string
        with self.assertRaises(TypeError):
```

(suite sur la page suivante)

(suite de la page précédente)

```
            s.split(2)

if __name__ == '__main__':
    unittest.main()
```

Un scénario de test est créé comme classe-fille de `unittest.TestCase`. Les trois tests individuels sont définis par des méthodes dont les noms commencent par les lettres `test`. Cette convention de nommage signale au lanceur de tests quelles méthodes sont des tests.

Le cœur de chaque test est un appel à `assertEqual()` pour vérifier un résultat attendu ; `assertTrue()` ou `assertFalse()` pour vérifier une condition ; ou `assertRaises()` pour vérifier qu'une exception particulière est levée. Ces méthodes sont utilisées à la place du mot-clé `assert` pour que le lanceur de tests puisse récupérer les résultats de tous les tests et produire un rapport.

Les méthodes `setUp()` et `tearDown()` vous autorisent à définir des instructions qui seront exécutées avant et après chaque méthode test. Elles sont davantage détaillées dans la section *Organiser le code de test*.

Le bloc final montre une manière simple de lancer les tests. `unittest.main()` fournit une interface en ligne de commande pour le script de test. Lorsqu'il est lancé en ligne de commande, le script ci-dessus produit une sortie qui ressemble à ceci :

```
...
----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

Passer l'option `-v` à votre script de test informera `unittest.main()` qu'il doit fournir un niveau plus important de détails, et produit la sortie suivante :

```
test_isupper (__main__.TestStringMethods.test_isupper) ... ok
test_split (__main__.TestStringMethods.test_split) ... ok
test_upper (__main__.TestStringMethods.test_upper) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.001s

OK
```

Les exemples ci-dessus montrent les fonctionnalités d'`unittest` les plus communément utilisées et qui sont suffisantes pour couvrir les besoins courants en matière de test. Le reste de la documentation explore l'ensemble complet des fonctionnalités depuis les premiers principes.

Modifié dans la version 3.11 : The behavior of returning a value from a test method (other than the default `None` value), is now deprecated.

27.5.2 Interface en ligne de commande

Le module *unittest* est utilisable depuis la ligne de commande pour exécuter des tests à partir de modules, de classes ou même de méthodes de test individuelles :

```
python -m unittest test_module1 test_module2
python -m unittest test_module.TestClass
python -m unittest test_module.TestClass.test_method
```

La commande accepte en argument une liste de n'importe quelle combinaison de noms de modules et de noms de classes ou de méthodes entièrement qualifiés.

Les modules de test peuvent également être spécifiés par un chemin de fichier :

```
python -m unittest tests/test_something.py
```

Cette fonctionnalité permet d'utiliser la complétion de l'interpréteur de commandes système (*le shell*) pour spécifier le module de test. Le chemin est converti en nom de module en supprimant le `.py` et en convertissant les séparateurs de chemin en '.'. Si vous voulez exécuter un fichier test qui n'est pas importable en tant que module, exécutez directement le fichier.

Pour obtenir plus de détails lors de l'exécution utilisez l'option `-v` (plus de verbosité) :

```
python -m unittest -v test_module
```

Quand la commande est exécutée sans arguments *Découverte des tests* est lancée :

```
python -m unittest
```

Pour afficher la liste de toutes les options de la commande utilisez l'option `-h` :

```
python -m unittest -h
```

Modifié dans la version 3.2 : Dans les versions antérieures, il était seulement possible d'exécuter des méthodes de test individuelles et non des modules ou des classes.

Ajouté dans la version 3.14 : Output is colorized by default and can be controlled using environment variables.

Options de la ligne de commande

Le programme **`unittest`** gère ces options de la ligne de commande :

`-b, --buffer`

Les flux de sortie et d'erreur standards sont mis en mémoire tampon pendant l'exécution des tests. L'affichage produit par un test réussi n'est pas pris en compte. Les sorties d'affichages d'un test en échec ou en erreur sont conservés et ajoutés aux messages d'erreur.

`-c, --catch`

Utiliser `Control-C` pendant l'exécution des tests attend que le test en cours se termine, puis affiche tous les résultats obtenus jusqu'ici. Une seconde utilisation de `Control-C` provoque l'exception normale `KeyboardInterrupt`.

Voir *Signal Handling* pour les fonctions qui utilisent cette fonctionnalité.

`-f, --failfast`

Arrête l'exécution des tests lors du premier cas d'erreur ou d'échec.

`-k`

Only run test methods and classes that match the pattern or substring. This option may be used multiple times, in which case all test cases that match any of the given patterns are included.

Les motifs qui contiennent un caractère de remplacement (`*`) sont comparés au nom du test en utilisant `fnmatch.fnmatchcase()` ; sinon, une recherche simple de sous chaîne respectant la casse est faite.

Les motifs sont comparés au nom de la méthode de test complètement qualifiée tel qu'importé par le chargeur de test.

Par exemple, `-k machin` retient les tests `machin_tests.UnTest.test_untruc`, `truc_tests.UnTest.test_machin`, mais pas `truc_tests.MachinTest.test_untruc`.

`--locals`

Affiche les variables locales dans les traces d'appels.

`--durations N`

Show the N slowest test cases (N=0 for all).

Ajouté dans la version 3.2 : Les options de ligne de commande `-b`, `-c` et `-f` ont été ajoutées.

Ajouté dans la version 3.5 : Ajout de l'option de ligne de commande `--locals`.

Ajouté dans la version 3.7 : Ajout de l'option de ligne de commande `-k`.

Ajouté dans la version 3.12 : The command-line option `--durations`.

La ligne de commande peut également être utilisée pour découvrir les tests, pour exécuter tous les tests dans un projet ou juste un sous-ensemble.

27.5.3 Découverte des tests

Ajouté dans la version 3.2.

Unittest supports simple test discovery. In order to be compatible with test discovery, all of the test files must be modules or packages importable from the top-level directory of the project (this means that their filenames must be valid identifiers).

La découverte de tests est implémentée dans *`TestLoader.discover()`*, mais peut également être utilisée depuis la ligne de commande. Par exemple :

```
cd project_directory
python -m unittest discover
```

> **Note**
>
> Comme raccourci, `python -m unittest` est l'équivalent de `python -m unittest discover`. Pour passer des arguments au système de découverte des tests, la sous-commande `discover` doit être utilisée explicitement.

La sous-commande `discover` a les options suivantes :

`-v, --verbose`
: Affichage plus détaillé

`-s, --start-directory` `directory`
: Répertoire racine pour démarrer la découverte (`.` par défaut).

`-p, --pattern` `pattern`
: Motif de détection des fichiers de test (`test*.py` par défaut)

`-t, --top-level-directory` `directory`
: Dossier du premier niveau du projet (Par défaut le dossier de départ)

Les options *`-s`*, *`-p`* et *`-t`* peuvent être passées en arguments positionnels dans cet ordre. Les deux lignes de commande suivantes sont équivalentes :

```
python -m unittest discover -s project_directory -p "*_test.py"
python -m unittest discover project_directory "*_test.py"
```

Il est aussi possible de passer un nom de paquet plutôt qu'un chemin, par exemple `monprojet.souspaquet.test`, comme répertoire racine. Le nom du paquet fourni est alors importé et son emplacement sur le système de fichiers est utilisé comme répertoire racine.

> **Prudence**
>
> Le mécanisme de découverte charge les tests en les important. Une fois que le système a trouvé tous les fichiers de tests du répertoire de démarrage spécifié, il transforme les chemins en noms de paquets à importer. Par exemple `truc/bidule/machin.py` est importé sous `truc.bidule.machin`.

> Si un paquet est installé globalement et que le mécanisme de découverte de tests est effectué sur une copie différente du paquet, l'importation *peut* se produire à partir du mauvais endroit. Si cela arrive, le système émet un avertissement et se termine.
>
> Si vous donnez le répertoire racine sous la forme d'un nom de paquet plutôt que d'un chemin d'accès à un répertoire, alors Python suppose que l'emplacement à partir duquel il importe est l'emplacement que vous voulez, vous ne verrez donc pas l'avertissement.

Les modules de test et les paquets peuvent adapter le chargement et la découverte des tests en utilisant le protocole *load_tests protocol*.

Modifié dans la version 3.4 : Test discovery supports *namespace packages*.

Modifié dans la version 3.11 : Test discovery dropped the *namespace packages* support. It has been broken since Python 3.7. Start directory and its subdirectories containing tests must be regular package that have `__init__.py` file.

If the start directory is the dotted name of the package, the ancestor packages can be namespace packages.

Modifié dans la version 3.14 : Test discovery supports *namespace package* as start directory again. To avoid scanning directories unrelated to Python, tests are not searched in subdirectories that do not contain `__init__.py`.

27.5.4 Organiser le code de test

Les éléments de base des tests unitaires sont les *scénarios de tests* (*test cases* en anglais) --- Des scénarios uniques qui sont mis en place et exécutés pour vérifier qu'ils sont corrects. Dans `unittest`, les scénarios de test sont représentés par des instances de `unittest.TestCase`. Pour créer vos propres scénarios de test, vous devez écrire des sous-classes de `TestCase` ou utiliser `FunctionTestCase`.

Le code de test d'une instance de `TestCase` doit être entièrement autonome, de sorte qu'il puisse être exécuté soit de manière isolée, soit en combinaison arbitraire avec un nombre quelconque d'autres scénarios de test.

La sous-classe `TestCase` la plus simple va tout simplement implémenter une méthode de test (c'est-à-dire une méthode dont le nom commence par `test`) afin d'exécuter un code de test spécifique :

```
import unittest

class DefaultWidgetSizeTestCase(unittest.TestCase):
    def test_default_widget_size(self):
        widget = Widget('The widget')
        self.assertEqual(widget.size(), (50, 50))
```

Note that in order to test something, we use one of the *assert* methods* provided by the `TestCase` base class. If the test fails, an exception will be raised with an explanatory message, and `unittest` will identify the test case as a *failure*. Any other exceptions will be treated as *errors*.

Les tests peuvent être nombreux et leur mise en place peut être répétitive. Heureusement, on peut factoriser le code de mise en place en implémentant une méthode appelée `setUp()`, que le système de test appelle automatiquement pour chaque test exécuté :

```
import unittest

class WidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget('The widget')

    def test_default_widget_size(self):
        self.assertEqual(self.widget.size(), (50,50),
                         'incorrect default size')

    def test_widget_resize(self):
```

(suite sur la page suivante)

(suite de la page précédente)

```
        self.widget.resize(100,150)
        self.assertEqual(self.widget.size(), (100,150),
                         'wrong size after resize')
```

> **Note**
>
> L'ordre dans lequel les différents tests sont exécutés est déterminé en classant les noms des méthodes de test en fonction de la relation d'ordre des chaines de caractères .

Si la méthode `setUp()` lève une exception pendant l'exécution du test, le système considère que le test a subi une erreur, et la méthode test n'est pas exécutée.

De même, on peut fournir une méthode `tearDown()` qui nettoie après l'exécution de la méthode de test :

```
import unittest

class WidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget('The widget')

    def tearDown(self):
        self.widget.dispose()
```

Si `setUp()` a réussi, `tearDown()` est exécutée, que la méthode de test ait réussi ou non.

Un tel environnement de travail pour le code de test s'appelle un *aménagement de test* (*fixture* en anglais). Une nouvelle instance de *TestCase* est créée sous la forme d'un dispositif de test unique utilisé pour exécuter chaque méthode de test individuelle. Ainsi `setUp()`, `tearDown()` et `__init__()` ne sont appelées qu'une fois par test.

Il est recommandé d'utiliser *TestCase* pour regrouper les tests en fonction des fonctionnalités qu'ils testent. `unittest` fournit un mécanisme pour cela : la *suite de tests*, représentée par `TestSuite` du module `unittest`. Dans la plupart des cas, appeler `unittest.main()` fait correctement les choses et trouve tous les scénarios de test du module pour vous et les exécute.

Cependant, si vous voulez personnaliser la construction de votre suite de tests, vous pouvez le faire vous-même :

```
def suite():
    suite = unittest.TestSuite()
    suite.addTest(WidgetTestCase('test_default_widget_size'))
    suite.addTest(WidgetTestCase('test_widget_resize'))
    return suite

if __name__ == '__main__':
    runner = unittest.TextTestRunner()
    runner.run(suite())
```

Vous pouvez placer les définitions des scénarios de test et des suites de test dans le même module que le code à tester (tel que `composant.py`), mais il y a plusieurs avantages à placer le code de test dans un module séparé, tel que `test_composant.py` :

- Le module de test peut être exécuté indépendamment depuis la ligne de commande.
- Le code de test est plus facilement séparable du code livré.
- La tentation est moins grande de changer le code de test pour l'adapter au code qu'il teste sans avoir une bonne raison.
- Le code de test doit être modifié beaucoup moins souvent que le code qu'il teste.
- Le code testé peut être réusiné plus facilement.
- Les tests pour les modules écrits en C doivent de toute façon être dans des modules séparés, alors pourquoi ne pas être cohérent ?
- Si la stratégie de test change, il n'est pas nécessaire de changer le code source.

27.5.5 Réutilisation d'ancien code de test

Certains utilisateurs constatent qu'ils ont du code de test existant qu'ils souhaitent exécuter à partir de `unittest`, sans convertir chaque ancienne fonction de test en une sous-classe de `TestCase`.

Pour cette raison, `unittest` fournit une classe `FunctionTestCase`. Cette sous-classe de `TestCase` peut être utilisée pour encapsuler une fonction de test existante. Des fonctions de mise en place (*setUp*) et de démantèlement (*tearDown*) peuvent également être fournies.

Étant donnée la fonction de test suivante :

```
def testSomething():
    something = makeSomething()
    assert something.name is not None
    # ...
```

on peut créer une instance de scénario de test équivalente, avec des méthodes optionnelles de mise en place et de démantèlement :

```
testcase = unittest.FunctionTestCase(testSomething,
                                     setUp=makeSomethingDB,
                                     tearDown=deleteSomethingDB)
```

Note

Même si la classe `FunctionTestCase` peut être utilisée pour convertir rapidement une base de test existante vers un système basé sur `unittest`, cette approche n'est pas recommandée. Prendre le temps de bien configurer les sous-classes de `TestCase` simplifiera considérablement les futurs réusinages des tests.

Dans certains cas, les tests déjà existants ont pu être écrits avec le module `doctest`. Dans ce cas, `doctest` fournit une classe `DocTestSuite` qui peut construire automatiquement des instances de la classe `unittest.TestSuite` depuis des tests basés sur le module `doctest`.

27.5.6 Ignorer des tests et des erreurs prévisibles

Ajouté dans la version 3.1.

Unittest permet d'ignorer des méthodes de test individuelles et même des classes entières de tests. De plus, il prend en charge le marquage d'un test comme étant une "erreur prévue". Un test qui est cassé et qui échoue, mais qui ne doit pas être considéré comme un échec dans la classe `TestResult`.

Ignorer un test consiste à soit utiliser le *décorateur* `skip()` ou une de ses variantes conditionnelles, soit appeler `TestCase.skipTest()` à l'intérieur d'une méthode `setUp()` ou de test, soit lever `SkipTest` directement.

Un exemple de tests à ignorer :

```
class MyTestCase(unittest.TestCase):

    @unittest.skip("demonstrating skipping")
    def test_nothing(self):
        self.fail("shouldn't happen")

    @unittest.skipIf(mylib.__version__ < (1, 3),
                     "not supported in this library version")
    def test_format(self):
        # Tests that work for only a certain version of the library.
        pass

    @unittest.skipUnless(sys.platform.startswith("win"), "requires Windows")
```

(suite sur la page suivante)

(suite de la page précédente)

```
    def test_windows_support(self):
        # windows specific testing code
        pass

    def test_maybe_skipped(self):
        if not external_resource_available():
            self.skipTest("external resource not available")
        # test code that depends on the external resource
        pass
```

Ceci est le résultat de l'exécution de l'exemple ci-dessus en mode verbeux :

```
test_format (__main__.MyTestCase.test_format) ... skipped 'not supported in this␣
↪library version'
test_nothing (__main__.MyTestCase.test_nothing) ... skipped 'demonstrating skipping
↪'
test_maybe_skipped (__main__.MyTestCase.test_maybe_skipped) ... skipped 'external␣
↪resource not available'
test_windows_support (__main__.MyTestCase.test_windows_support) ... skipped
↪'requires Windows'

----------------------------------------------------------------------
Ran 4 tests in 0.005s

OK (skipped=4)
```

Les classes peuvent être ignorées tout comme les méthodes :

```
@unittest.skip("showing class skipping")
class MySkippedTestCase(unittest.TestCase):
    def test_not_run(self):
        pass
```

La méthode `TestCase.setUp()` permet également d'ignorer le test. Ceci est utile lorsqu'une ressource qui doit être configurée n'est pas disponible.

Les erreurs prévisibles utilisent le décorateur `expectedFailure()`

```
class ExpectedFailureTestCase(unittest.TestCase):
    @unittest.expectedFailure
    def test_fail(self):
        self.assertEqual(1, 0, "broken")
```

Il est facile de faire ses propres décorateurs en créant un décorateur qui appelle `skip()` sur le test que vous voulez ignorer. Par exemple, ce décorateur ignore le test à moins que l'objet passé ne possède un certain attribut :

```
def skipUnlessHasattr(obj, attr):
    if hasattr(obj, attr):
        return lambda func: func
    return unittest.skip("{!r} doesn't have {!r}".format(obj, attr))
```

Les décorateurs et exceptions suivants implémentent le système d'omission des tests et les erreurs prévisibles :

`@unittest.skip(reason)`

Ignore sans condition le test décoré. *La raison* doit décrire la raison pour laquelle le test est omis.

`@unittest.skipIf(condition, reason)`

Ignore le test décoré si la *condition* est vraie.

`@unittest.skipUnless(condition, reason)`

Ignore le test décoré sauf si la *condition* est vraie.

`@unittest.expectedFailure`

Mark the test as an expected failure or error. If the test fails or errors in the test function itself (rather than in one of the *test fixture* methods) then it will be considered a success. If the test passes, it will be considered a failure.

exception `unittest.SkipTest(reason)`

Cette exception est levée pour ignorer un test.

Habituellement, on utilise `TestCase.skipTest()` ou l'un des décorateurs d'omission au lieu de le lever une exception directement.

Les tests ignorés ne lancent ni `setUp()` ni `tearDown()`. Les classes ignorées ne lancent ni `setUpClass()` ni `tearDownClass()`. Les modules sautés n'ont pas `setUpModule()` ou `tearDownModule()` d'exécutés.

27.5.7 Distinguer les itérations de test à l'aide de sous-tests

Ajouté dans la version 3.4.

Lorsque certains de vos tests ne diffèrent que par de très petites différences, par exemple certains paramètres, *unittest* vous permet de les distinguer en utilisant le gestionnaire de contexte `subTest()` dans le corps d'une méthode de test.

Par exemple, le test suivant :

```
class NumbersTest(unittest.TestCase):

    def test_even(self):
        """
        Test that numbers between 0 and 5 are all even.
        """
        for i in range(0, 6):
            with self.subTest(i=i):
                self.assertEqual(i % 2, 0)
```

produit le résultat suivant :

```
======================================================================
FAIL: test_even (__main__.NumbersTest.test_even) (i=1)
Test that numbers between 0 and 5 are all even.
----------------------------------------------------------------------
Traceback (most recent call last):
  File "subtests.py", line 11, in test_even
    self.assertEqual(i % 2, 0)
    ^^^^^^^^^^^^^^^^^^^^^^^^^^
AssertionError: 1 != 0

======================================================================
FAIL: test_even (__main__.NumbersTest.test_even) (i=3)
Test that numbers between 0 and 5 are all even.
----------------------------------------------------------------------
Traceback (most recent call last):
  File "subtests.py", line 11, in test_even
    self.assertEqual(i % 2, 0)
    ^^^^^^^^^^^^^^^^^^^^^^^^^^
AssertionError: 1 != 0

======================================================================
```

(suite sur la page suivante)

(suite de la page précédente)

```
FAIL: test_even (__main__.NumbersTest.test_even) (i=5)
Test that numbers between 0 and 5 are all even.
----------------------------------------------------------------------
Traceback (most recent call last):
  File "subtests.py", line 11, in test_even
    self.assertEqual(i % 2, 0)
    ^^^^^^^^^^^^^^^^^^^^^^^^^^
AssertionError: 1 != 0
```

Sans l'utilisation d'un sous-test, l'exécution se termine après le premier échec, et l'erreur est moins facile à diagnostiquer car la valeur de `i` ne s'affiche pas :

```
======================================================================
FAIL: test_even (__main__.NumbersTest.test_even)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "subtests.py", line 32, in test_even
    self.assertEqual(i % 2, 0)
AssertionError: 1 != 0
```

27.5.8 Classes et fonctions

Cette section décrit en détail l'API de `unittest`.

Scénarios de tests

class unittest.**TestCase**(*methodName='runTest'*)

Les instances de la classe `TestCase` représentent des tests logiques unitaires dans l'univers `unittest`. Cette classe est conçue pour être utilisée comme classe de base. Les scénarios de tests sont à implémenter en héritant de cette classe. La classe implémente l'interface nécessaire au lanceur de tests pour lui permettre de les exécuter ainsi que les méthodes que le code de test peut utiliser pour vérifier et signaler les différents types d'erreurs.

Chaque instance de la classe `TestCase` n'exécute qu'une seule méthode de base : la méthode nommée *methodName* . Dans la plupart des utilisations de la classe `TestCase`, vous n'avez pas à changer le nom de la méthode, ni à réimplémenter la méthode `runTest()`.

Modifié dans la version 3.2 : La classe `TestCase` peut désormais être utilisée sans passer de paramètre *methodName*. Cela facilite l'usage de `TestCase` dans l'interpréteur interactif.

Les instances de la classe `TestCase` fournissent trois groupes de méthodes : un groupe utilisé pour exécuter le test, un autre utilisé par l'implémentation du test pour vérifier les conditions et signaler les échecs, et quelques méthodes de recherche permettant de recueillir des informations sur le test lui-même.

Les méthodes du premier groupe (exécution du test) sont :

setUp()

Méthode appelée pour réaliser la mise en place du test. Elle est exécutée immédiatement avant l'appel de la méthode de test ; à l'exception de `AssertionError` ou `SkipTest`, toute exception levée par cette méthode est considérée comme une erreur et non pas comme un échec du test. L'implémentation par défaut ne fait rien.

tearDown()

Méthode appelée immédiatement après l'appel de la méthode de test et l'enregistrement du résultat. Elle est appelée même si la méthode de test a levé une exception. De fait, l'implémentation d'un sous-classes doit être fait avec précaution si vous vérifiez l'état interne de la classe. Toute exception, autre que `AssertionError` ou `SkipTest`, levée par cette méthode est considérée comme une erreur supplémentaire plutôt que comme un échec du test (augmentant ainsi le nombre total des erreurs signalées). Cette méthode est appelée uniquement si l'exécution de `setUp()` est réussie quel que soit le résultat de la méthode de test. L'implémentation par défaut ne fait rien.

setUpClass()

Méthode de classe appelée avant l'exécution des tests dans la classe en question. `setUpClass` est appelée avec la classe comme seul argument et doit être décorée comme une `classmethod()` :

```
@classmethod
def setUpClass(cls):
    ...
```

Voir *Class and Module Fixtures* pour plus de détails.

Ajouté dans la version 3.2.

tearDownClass()

Méthode de classe appelée après l'exécution des tests de la classe en question. `tearDownClass` est appelée avec la classe comme seul argument et doit être décorée comme une `classmethod()` :

```
@classmethod
def tearDownClass(cls):
    ...
```

Voir *Class and Module Fixtures* pour plus de détails.

Ajouté dans la version 3.2.

run(*result=None*)

Exécute le test, en collectant le résultat dans l'objet `TestResult` passé comme *result*. Si *result* est omis ou vaut `None`, un objet temporaire de résultat est créé (en appelant la méthode `defaultTestResult()`) et utilisé. L'objet résultat est renvoyé à l'appelant de `run()`.

Le même effet peut être obtenu en appelant simplement l'instance `TestCase`.

Modifié dans la version 3.3 : Les versions précédentes de `run` ne renvoyaient pas le résultat. Pas plus que l'appel d'une instance.

skipTest(*reason*)

Appeler cette fonction pendant l'exécution d'une méthode de test ou de `setUp()` permet d'ignorer le test en cours. Voir *Ignorer des tests et des erreurs prévisibles* pour plus d'informations.

Ajouté dans la version 3.1.

subTest(*msg=None*, ***params*)

Renvoie un gestionnaire de contexte qui exécute le bloc de code du contexte comme un sous-test. *msg* et *params* sont des valeurs optionnelles et arbitraires qui sont affichées chaque fois qu'un sous-test échoue, permettant de les identifier clairement.

Un scénario de test peut contenir un nombre quelconque de déclarations de sous-test, et elles peuvent être imbriquées librement.

Voir *Distinguer les itérations de test à l'aide de sous-tests* pour plus d'informations.

Ajouté dans la version 3.4.

debug()

Lance le test sans collecter le résultat. Ceci permet aux exceptions levées par le test d'être propagées à l'appelant, et donc peut être utilisé pour exécuter des tests sous un débogueur.

La classe `TestCase` fournit plusieurs méthodes d'assertion pour vérifier et signaler les échecs. Le tableau suivant énumère les méthodes les plus couramment utilisées (voir les tableaux ci-dessous pour plus de méthodes d'assertion) :

Méthode	Vérifie que	Disponible en
`assertEqual(a, b)`	`a == b`	
`assertNotEqual(a, b)`	`a != b`	
`assertTrue(x)`	`bool(x) is True`	
`assertFalse(x)`	`bool(x) is False`	
`assertIs(a, b)`	`a is b`	3.1
`assertIsNot(a, b)`	`a is not b`	3.1
`assertIsNone(x)`	`x is None`	3.1
`assertIsNotNone(x)`	`x is not None`	3.1
`assertIn(a, b)`	`a in b`	3.1
`assertNotIn(a, b)`	`a not in b`	3.1
`assertIsInstance(a, b)`	`isinstance(a, b)`	3.2
`assertNotIsInstance(a, b)`	`not isinstance(a, b)`	3.2
`assertIsSubclass(a, b)`	`issubclass(a, b)`	3.14
`assertNotIsSubclass(a, b)`	`not issubclass(a, b)`	3.14

Toutes les méthodes *assert* prennent en charge un argument *msg* qui, s'il est spécifié, est utilisé comme message d'erreur en cas d'échec (voir aussi `longMessage`). Notez que l'argument nommé *msg* peut être passé à `assertRaises()`, `assertRaisesRegex()`, `assertWarns()`, `assertWarnsRegex()`, seulement quand elles sont utilisées comme gestionnaire de contexte.

assertEqual(*first*, *second*, *msg=None*)

Vérifie que *first* et *second* sont égaux. Si les valeurs ne sont pas égales, le test échouera.

En outre, si *first* et *second* ont exactement le même type et sont de type *list*, *tuple*, *dict*, *set*, *frozenset* ou *str* ou tout autre type de sous classe enregistrée dans `addTypeEqualityFunc()`. La fonction égalité spécifique au type sera appelée pour générer une erreur plus utile (voir aussi *liste des méthodes spécifiques de type*).

Modifié dans la version 3.1 : Ajout de l'appel automatique de la fonction d'égalité spécifique au type.

Modifié dans la version 3.2 : Ajout de `assertMultiLineEqual()` comme fonction d'égalité de type par défaut pour comparer les chaînes.

assertNotEqual(*first*, *second*, *msg=None*)

Vérifie que *first* et *second* ne sont pas égaux. Si les valeurs sont égales, le test échouera.

assertTrue(*expr*, *msg=None*)

assertFalse(*expr*, *msg=None*)

Vérifie que *expr* est vraie (ou fausse).

Notez que cela revient à utiliser `bool(expr) is True` et non à `expr is True` (utilisez `assertIs(expr, True)` pour cette dernière). Cette méthode doit également être évitée lorsque des méthodes plus spécifiques sont disponibles (par exemple `assertEqual(a, b)` au lieu de `assertTrue(a == b)`), car elles fournissent un meilleur message d'erreur en cas d' échec.

assertIs(*first*, *second*, *msg=None*)

assertIsNot(*first*, *second*, *msg=None*)

Vérifie que *first* et *second* sont (ou ne sont pas) le même objet.

Ajouté dans la version 3.1.

assertIsNone(*expr*, *msg=None*)

assertIsNotNone(*expr*, *msg=None*)

Vérifie que *expr* est (ou n'est pas) la valeur `None`.

Ajouté dans la version 3.1.

assertIn(*member*, *container*, *msg=None*)

assertNotIn(*member*, *container*, *msg=None*)

Vérifie que *member* est (ou n'est pas) dans *container*.

Ajouté dans la version 3.1.

assertIsInstance(*obj*, *cls*, *msg=None*)

assertNotIsInstance(*obj*, *cls*, *msg=None*)

Vérifie que *obj* est (ou n'est pas) une instance de *cls* (Ce qui peut être une classe ou un *n*-uplet de classes, comme utilisée par `isinstance()`). Pour vérifier le type exact, utilisez `assertIs(type(obj), cls)`.

Ajouté dans la version 3.2.

assertIsSubclass(*cls*, *superclass*, *msg=None*)

assertNotIsSubclass(*cls*, *superclass*, *msg=None*)

Test that *cls* is (or is not) a subclass of *superclass* (which can be a class or a tuple of classes, as supported by `issubclass()`). To check for the exact type, use `assertIs(cls, superclass)`.

Ajouté dans la version 3.14.

Il est également possible de vérifier la production des exceptions, des avertissements et des messages de journaux à l'aide des méthodes suivantes :

Méthode	Vérifie que	Disponible en
`assertRaises(exc, fun, *args, **kwds)`	`fun(*args, **kwds)` lève bien l'exception *exc*	
`assertRaisesRegex(exc, r, fun, *args, **kwds)`	`fun(*args, **kwds)` lève bien l'exception *exc* et que le message correspond au motif de l'expression régulière *r*	3.1
`assertWarns(warn, fun, *args, **kwds)`	`fun(*args, **kwds)` lève bien l'avertissement *warn*	3.2
`assertWarnsRegex(warn, r, fun, *args, **kwds)`	`fun(*args, **kwds)` lève bien l'avertissement *warn* et que le message correspond au motif de l'expression régulière *r*	3.2
`assertLogs(logger, level)`	Le bloc `with` écrit dans le *logger* avec un niveau minimum égal à *level*	3.4
`assertNoLogs(logger, level)`	**The `with` block does not log on** *logger* with minimum *level*	3.10

assertRaises(*exception*, *callable*, **args*, ***kwds*)

assertRaises(*exception*, ***, *msg=None*)

Vérifie qu'une exception est levée lorsque *callable* est appelé avec n'importe quel argument positionnel ou nommé qui est également passé à `assertRaises()`. Le test réussit si *exception* est levée, est en erreur si une autre exception est levée, ou en échec si aucune exception n'est levée. Pour capturer une exception d'un groupe d'exceptions, un *n*-uplet contenant les classes d'exceptions peut être passé à *exception*.

Si seuls les arguments *exception* et éventuellement *msg* sont donnés, renvoie un gestionnaire de contexte pour que le code sous test puisse être écrit en ligne plutôt que comme une fonction :

```
with self.assertRaises(SomeException):
    do_something()
```

Lorsqu'il est utilisé comme gestionnaire de contexte, `assertRaises()` accepte l'argument nommé supplémentaire *msg*.

Le gestionnaire de contexte enregistre l'exception interceptée dans son attribut `exception`. Ceci est particulièrement utile si l'intention est d'effectuer des contrôles supplémentaires sur l'exception levée :

```
with self.assertRaises(SomeException) as cm:
    do_something()

the_exception = cm.exception
self.assertEqual(the_exception.error_code, 3)
```

Modifié dans la version 3.1 : Ajout de la possibilité d'utiliser `assertRaises()` comme gestionnaire de contexte.

Modifié dans la version 3.2 : Ajout de l'attribut `exception`.

Modifié dans la version 3.3 : Ajout de l'argument nommé *msg* lorsqu'il est utilisé comme gestionnaire de contexte.

assertRaisesRegex(*exception*, *regex*, *callable*, **args*, ***kwds*)

assertRaisesRegex(*exception*, *regex*, ***, *msg=None*)

Comme `assertRaises()` mais vérifie aussi que *regex* correspond à la représentation de la chaîne de caractères de l'exception levée. *regex* peut être un objet d'expression rationnelle ou une chaîne contenant une expression rationnelle appropriée pour être utilisée par `re.search()`. Exemples :

```
self.assertRaisesRegex(ValueError, "invalid literal for.*XYZ'$",
                       int, 'XYZ')
```

ou :

```
with self.assertRaisesRegex(ValueError, 'literal'):
   int('XYZ')
```

Ajouté dans la version 3.1 : Sous le nom `assertRaisesRegexp`.

Modifié dans la version 3.2 : Renommé en `assertRaisesRegex()`.

Modifié dans la version 3.3 : Ajout de l'argument nommé *msg* lorsqu'il est utilisé comme gestionnaire de contexte.

assertWarns(*warning*, *callable*, **args*, ***kwds*)

assertWarns(*warning*, ***, *msg=None*)

Test qu'un avertissement est déclenché lorsque *callable* est appelé avec n'importe quel argument positionnel ou nommé qui est également passé à `assertWarns()`. Le test passe si *warning* est déclenché et échoue s'il ne l'est pas. Toute exception est une erreur. Pour capturer un avertissement dans un ensemble d'avertissements, un *n*-uplet contenant les classes d'avertissement peut être passé à *warnings*.

Si seuls les arguments * warning* et éventuellement *msg* sont donnés, renvoie un gestionnaire de contexte pour que le code testé puisse être écrit en ligne plutôt que comme une fonction :

```
with self.assertWarns(SomeWarning):
    do_something()
```

Lorsqu'il est utilisé comme gestionnaire de contexte, `assertWarns()` accepte l'argument nommé supplémentaire *msg*.

Le gestionnaire de contexte stocke l'avertissement capturé dans son attribut `warning`, et la ligne source qui a déclenché les avertissements dans les attributs `filename` et `lineno`. Cette fonction peut être utile si l'intention est d'effectuer des contrôles supplémentaires sur l'avertissement capturé :

```
with self.assertWarns(SomeWarning) as cm:
    do_something()
```

(suite sur la page suivante)

(suite de la page précédente)

```
self.assertIn('myfile.py', cm.filename)
self.assertEqual(320, cm.lineno)
```

Cette méthode fonctionne indépendamment des filtres d'avertissement en place lorsqu'elle est appelée.

Ajouté dans la version 3.2.

Modifié dans la version 3.3 : Ajout de l'argument nommé *msg* lorsqu'il est utilisé comme gestionnaire de contexte.

assertWarnsRegex(*warning*, *regex*, *callable*, **args*, ***kwds*)

assertWarnsRegex(*warning*, *regex*, ***, *msg=None*)

Comme `assertWarns()` mais vérifie aussi qu'une *regex* corresponde au message de l'avertissement. *regex* peut être un objet d'expression régulière ou une chaîne contenant une expression régulière appropriée pour être utilisée par `re.search()`. Exemple :

```
self.assertWarnsRegex(DeprecationWarning,
                      r'legacy_function\(\) is deprecated',
                      legacy_function, 'XYZ')
```

ou :

```
with self.assertWarnsRegex(RuntimeWarning, 'unsafe frobnicating'):
    frobnicate('/etc/passwd')
```

Ajouté dans la version 3.2.

Modifié dans la version 3.3 : Ajout de l'argument nommé *msg* lorsqu'il est utilisé comme gestionnaire de contexte.

assertLogs(*logger=None*, *level=None*)

Un gestionnaire de contexte pour tester qu'au moins un message est enregistré sur le *logger* ou un de ses enfants, avec au moins le *niveau* donné.

Si donné, *logger* doit être une classe `logging.logger` objet ou une classe `str` donnant le nom d'un journal. La valeur par défaut est le journal racine *root*, qui capture tous les messages qui n'ont pas été arrêtés par un *logger* ne propageant pas les messages.

If given, *level* should be either a numeric logging level or its string equivalent (for example either `"ERROR"` or `logging.ERROR`). The default is `logging.INFO`.

Le test passe si au moins un message émis à l'intérieur du bloc `with` correspond aux conditions *logger* et *level*, sinon il échoue.

L'objet retourné par le gestionnaire de contexte est une aide à l'enregistrement qui garde la trace des messages de journal correspondants. Il a deux attributs :

records

Une liste d'objets `logging.LogRecord` de messages de log correspondants.

output

Une liste d'objets `str` avec la sortie formatée des messages correspondants.

Exemple :

```
with self.assertLogs('foo', level='INFO') as cm:
    logging.getLogger('foo').info('first message')
    logging.getLogger('foo.bar').error('second message')
self.assertEqual(cm.output, ['INFO:foo:first message',
                             'ERROR:foo.bar:second message'])
```

Ajouté dans la version 3.4.

assertNoLogs (*logger=None*, *level=None*)

A context manager to test that no messages are logged on the *logger* or one of its children, with at least the given *level*.

If given, *logger* should be a `logging.Logger` object or a `str` giving the name of a logger. The default is the root logger, which will catch all messages.

If given, *level* should be either a numeric logging level or its string equivalent (for example either `"ERROR"` or `logging.ERROR`). The default is `logging.INFO`.

Unlike `assertLogs()`, nothing will be returned by the context manager.

Ajouté dans la version 3.10.

Il existe également d'autres méthodes utilisées pour effectuer des contrôles plus spécifiques, telles que :

Méthode	Vérifie que	Disponible en
`assertAlmostEqual(a, b)`	`round(a-b, 7) == 0`	
`assertNotAlmostEqual(a, b)`	`round(a-b, 7) != 0`	
`assertGreater(a, b)`	`a > b`	3.1
`assertGreaterEqual(a, b)`	`a >= b`	3.1
`assertLess(a, b)`	`a < b`	3.1
`assertLessEqual(a, b)`	`a <= b`	3.1
`assertRegex(s, r)`	`r.search(s)`	3.1
`assertNotRegex(s, r)`	`not r.search(s)`	3.2
`assertCountEqual(a, b)`	*a* et *b* ont les mêmes éléments dans le même nombre, quel que soit leur ordre.	3.2
`assertStartsWith(a, b)`	`a.startswith(b)`	3.14
`assertNotStartsWith(a, b)`	`not a.startswith(b)`	3.14
`assertEndsWith(a, b)`	`a.endswith(b)`	3.14
`assertNotEndsWith(a, b)`	`not a.endswith(b)`	3.14
`assertHasAttr(a, b)`	`hastattr(a, b)`	3.14
`assertNotHasAttr(a, b)`	`not hastattr(a, b)`	3.14

assertAlmostEqual (*first*, *second*, *places=7*, *msg=None*, *delta=None*)

assertNotAlmostEqual (*first*, *second*, *places=7*, *msg=None*, *delta=None*)

Vérifie que *first* et *second* sont approximativement (ou pas approximativement) égaux en calculant la différence, en arrondissant au nombre donné de décimales *places* (par défaut 7), et en comparant à zéro. Notez que ces méthodes arrondissent les valeurs au nombre donné de *décimales* (par exemple comme la fonction `round()`) et non aux *chiffres significatifs*.

Si *delta* est fourni au lieu de *places*, la différence entre *first* et *second* doit être inférieure ou égale (ou supérieure) à *delta*.

Fournir à la fois *delta* et *places* lève une `TypeError`.

Modifié dans la version 3.2 : `assertAlmostEqual()` considère automatiquement des objets presque égaux qui se comparent égaux. `assertNotNotAlmostEqual()` échoue automatiquement si les objets qui se comparent sont égaux. Ajout de l'argument nommé *delta*.

assertGreater (*first*, *second*, *msg=None*)

assertGreaterEqual (*first*, *second*, *msg=None*)

assertLess (*first*, *second*, *msg=None*)

assertLessEqual (*first*, *second*, *msg=None*)

Vérifie que *first* est respectivement >, >=, >=, < ou <= à *second* selon le nom de la méthode. Sinon, le test échouera :

```
>>> self.assertGreaterEqual(3, 4)
AssertionError: "3" unexpectedly not greater than or equal to "4"
```

Ajouté dans la version 3.1.

assertRegex (*text*, *regex*, *msg=None*)

assertNotRegex (*text*, *regex*, *msg=None*)

Vérifie qu'une recherche par motif *regex* correspond (ou ne correspond pas) à *text*. En cas d'échec, le message d'erreur inclura le motif et le *texte* (ou le motif et la partie du *texte* qui correspond de manière inattendue). *regex* peut être un objet d'expression régulière ou une chaîne contenant une expression régulière appropriée pour être utilisée par `re.search()`.

Ajouté dans la version 3.1 : Ajouté sous le nom `assertRegexpMatches`.

Modifié dans la version 3.2 : La méthode `assertRegexpMatches()` a été renommé en `assertRegex()`.

Ajouté dans la version 3.2 : `assertNotRegex()`.

assertCountEqual (*first*, *second*, *msg=None*)

Vérifie que la séquence *first* contient les mêmes éléments que *second*, quel que soit leur ordre. Si ce n'est pas le cas, un message d'erreur indiquant les différences entre les séquences est généré.

Les éléments en double ne sont *pas* ignorés lors de la comparaison entre *first* et *second*. Il vérifie si chaque élément a le même nombre dans les deux séquences. Équivalent à : `assertEqual(Counter(list(first)), Counter(list(second)))` mais fonctionne aussi avec des séquences d'objets non *hachables*.

Ajouté dans la version 3.2.

assertStartsWith (*s*, *prefix*, *msg=None*)

assertNotStartsWith (*s*, *prefix*, *msg=None*)

Test that the Unicode or byte string *s* starts (or does not start) with a *prefix*. *prefix* can also be a tuple of strings to try.

Ajouté dans la version 3.14.

assertEndsWith (*s*, *suffix*, *msg=None*)

assertNotEndsWith (*s*, *suffix*, *msg=None*)

Test that the Unicode or byte string *s* ends (or does not end) with a *suffix*. *suffix* can also be a tuple of strings to try.

Ajouté dans la version 3.14.

assertHasAttr (*obj*, *name*, *msg=None*)

assertNotHasAttr (*obj*, *name*, *msg=None*)

Test that the object *obj* has (or has not) an attribute *name*.

Ajouté dans la version 3.14.

La méthode `assertEqual()` envoie le contrôle d'égalité pour les objets du même type à différentes méthodes spécifiques au type. Ces méthodes sont déjà implémentées pour la plupart des types intégrés, mais il est également possible d'enregistrer de nouvelles méthodes en utilisant `addTypeEqualityFunc()` :

addTypeEqualityFunc (*typeobj*, *function*)

Enregistre une méthode spécifique appelée par `assertEqual()` pour vérifier si deux objets exactement

du même *typeobj* (et non leurs sous-classes) sont égaux. *function* doit prendre deux arguments positionnels et un troisième argument nommé *msg=None* tout comme `assertEqual()` le fait. Il doit lever `self.failureException(msg)` lorsqu'une inégalité entre les deux premiers paramètres est détectée en fournissant éventuellement des informations utiles et expliquant l'inégalité en détail dans le message d'erreur.

Ajouté dans la version 3.1.

La liste des méthodes spécifiques utilisées automatiquement par `assertEqual()` est résumée dans le tableau suivant. Notez qu'il n'est généralement pas nécessaire d'invoquer ces méthodes directement.

Méthode	Utilisé pour comparer	Disponible en
`assertMultiLineEqual(a, b)`	chaînes	3.1
`assertSequenceEqual(a, b)`	séquences	3.1
`assertListEqual(a, b)`	listes	3.1
`assertTupleEqual(a, b)`	*n*-uplets	3.1
`assertSetEqual(a, b)`	*sets* ou *frozensets*	3.1
`assertDictEqual(a, b)`	dictionnaires	3.1

assertMultiLineEqual(*first*, *second*, *msg=None*)

Vérifie que la chaîne sur plusieurs lignes *first* est égale à la chaîne *second*. Si les deux chaînes de caractères ne sont pas égales, un *diff* mettant en évidence les différences est inclus dans le message d'erreur. Cette méthode est utilisée par défaut pour comparer les chaînes avec `assertEqual()`.

Ajouté dans la version 3.1.

assertSequenceEqual(*first*, *second*, *msg=None*, *seq_type=None*)

Vérifie que deux séquences sont égales. Si un *seq_type* est fourni, *first* et *second* doivent tous deux être des instances de *seq_type* ou un échec est levé. Si les séquences sont différentes, un message d'erreur indiquant la différence entre les deux est généré.

Cette méthode n'est pas appelée directement par `assertEqual()`, mais sert à implémenter `assertListEqual()` et `assertTupleEqual()`.

Ajouté dans la version 3.1.

assertListEqual(*first*, *second*, *msg=None*)

assertTupleEqual(*first*, *second*, *msg=None*)

Vérifie que deux listes ou deux *n*-uplets sont égaux. Si ce n'est pas le cas, un message d'erreur qui ne montre que les différences entre les deux est généré. Une erreur est également signalée si l'un ou l'autre des paramètres n'est pas du bon type. Ces méthodes sont utilisées par défaut pour comparer des listes ou des *n*-uplets avec `assertEqual()`.

Ajouté dans la version 3.1.

assertSetEqual(*first*, *second*, *msg=None*)

Vérifie que deux ensembles sont égaux. Si ce n'est pas le cas, un message d'erreur s'affiche et indique les différences entre les *sets*. Cette méthode est utilisée par défaut lors de la comparaison de *sets* ou de *frozensets* avec `assertEqual()`.

Échoue si l'un des objets *first* ou *second* n'a pas de méthode `set.difference()`.

Ajouté dans la version 3.1.

assertDictEqual(*first*, *second*, *msg=None*)

Vérifie que deux dictionnaires sont égaux. Si ce n'est pas le cas, un message d'erreur qui montre les différences dans les dictionnaires est généré. Cette méthode est utilisée par défaut pour comparer les dictionnaires dans les appels à `assertEqual()`.

Ajouté dans la version 3.1.

Enfin, la classe `TestCase` fournit les méthodes et attributs suivants :

fail(*msg=None*)

Indique un échec du test sans condition, avec *msg* ou `None` pour le message d'erreur.

failureException

Cet attribut de classe donne l'exception levée par la méthode de test. Si un *framework* de tests doit utiliser une exception spécialisée, probablement pour enrichir l'exception d'informations additionnels., il doit hériter de cette classe d'exception pour *bien fonctionner* avec le *framework*. La valeur initiale de cet attribut est `AssertionError`.

longMessage

Cet attribut de classe détermine ce qui se passe lorsqu'un message d'échec personnalisé est passé en argument au paramètre *msg* à un appel *assertXYYY* qui échoue. `True` est la valeur par défaut. Dans ce cas, le message personnalisé est ajouté à la fin du message d'erreur standard. Lorsqu'il est réglé sur `False`, le message personnalisé remplace le message standard.

Le paramétrage de la classe peut être écrasé dans les méthodes de test individuelles en assignant un attribut d'instance, *self.longMessage*, à `True` ou `False` avant d'appeler les méthodes d'assertion.

Le réglage de la classe est réinitialisé avant chaque appel de test.

Ajouté dans la version 3.1.

maxDiff

Cet attribut contrôle la longueur maximale des *diffs* en sortie des méthodes qui génèrent des *diffs* en cas d'échec. La valeur par défaut est 80*8 caractères. Les méthodes d'assertions affectées par cet attribut sont `assertSequenceEqual()` (y compris toutes les méthodes de comparaison de séquences qui lui sont déléguées), `assertDictEqual()` et `assertMultiLineEqual()`.

Régler `maxDiff` sur `None` signifie qu'il n'y a pas de longueur maximale pour les *diffs*.

Ajouté dans la version 3.2.

Les *frameworks* de test peuvent utiliser les méthodes suivantes pour recueillir des informations sur le test :

countTestCases()

Renvoie le nombre de tests représentés par cet objet test. Pour les instances de `TestCase`, c'est toujours `1`.

defaultTestResult()

Retourne une instance de la classe de résultat de test qui doit être utilisée pour cette classe de cas de test (si aucune autre instance de résultat n'est fournie à la méthode `run()`).

Pour les instances de `TestCase`, c'est toujours une instance de `TestResult`; les sous-classes de `TestCase` peuvent la remplacer au besoin.

id()

Retourne une chaîne identifiant le cas de test spécifique. Il s'agit généralement du nom complet de la méthode de test, y compris le nom du module et de la classe.

shortDescription()

Renvoie une description du test, ou `None` si aucune description n'a été fournie. L'implémentation par défaut de cette méthode renvoie la première ligne de la *docstring* de la méthode de test, si disponible, ou `None`.

Modifié dans la version 3.1 : En 3.1, ceci a été modifié pour ajouter le nom du test à la description courte, même en présence d'une *docstring*. Cela a causé des problèmes de compatibilité avec les extensions *unittest* et l'ajout du nom du test a été déplacé dans la classe `TextTestResult` dans Python 3.2.

addCleanup(*function*, /, **args*, ***kwargs*)

Ajout d'une fonction à appeler après `tearDown()` pour nettoyer les ressources utilisées pendant le test. Les fonctions seront appelées dans l'ordre inverse de l'ordre dans lequel elles ont été ajoutées (LIFO). Elles sont appelées avec tous les arguments positionnels et arguments nommés passés à `addCleanup()` quand elles sont ajoutées.

Si `setUp()` échoue, cela signifie que `tearDown()` n'est pas appelé, alors que les fonctions de nettoyage

ajoutées seront toujours appelées.

Ajouté dans la version 3.1.

enterContext(*cm*)

Enter the supplied *context manager*. If successful, also add its `__exit__()` method as a cleanup function by `addCleanup()` and return the result of the `__enter__()` method.

Ajouté dans la version 3.11.

doCleanups()

Cette méthode est appelée sans conditions après `tearDown()`, ou après `setUp()` si `setUp()` lève une exception.

Cette méthode est chargée d'appeler toutes les fonctions de nettoyage ajoutées par `addCleanup()`. Si vous avez besoin de fonctions de nettoyage à appeler *avant* l'appel à `tearDown()` alors vous pouvez appeler `doCleanups()` vous-même.

`doCleanups()` extrait les méthodes de la pile des fonctions de nettoyage une à la fois, de sorte qu'elles peuvent être appelées à tout moment.

Ajouté dans la version 3.1.

classmethod addClassCleanup(*function*, /, **args*, ***kwargs*)

Ajout d'une fonction à appeler après `tearDownClass()` pour nettoyer les ressources utilisées par la classe test. Les fonctions sont appelées dans l'ordre inverse de l'ordre dans lequel elles ont été ajoutées (LIFO). Elles sont appelées avec tous les arguments positionnels et nommés passés à `addClassCleanup()` quand elles sont ajoutées.

Si `setUpClass()` échoue, impliquant que `tearDownClass()` n'est pas appelé, alors les fonctions de nettoyage ajoutées sont quand même appelées.

Ajouté dans la version 3.8.

classmethod enterClassContext(*cm*)

Enter the supplied *context manager*. If successful, also add its `__exit__()` method as a cleanup function by `addClassCleanup()` and return the result of the `__enter__()` method.

Ajouté dans la version 3.11.

classmethod doClassCleanups()

Cette méthode est appelée sans conditions après `tearDownClass()`, ou après `setUpClass()` si `setUpClass()` lève une exception.

Cette méthode est chargée d'appeler toutes les fonctions de nettoyage ajoutées par `addClassCleanup()`. Si vous avez besoin de fonctions de nettoyage à appeler *avant* l'appel à `tearDownClass()` alors vous pouvez appeler `doClassCleanups()` vous-même.

`doClassCleanups()` extrait les méthodes de la pile des fonctions de nettoyage une à la fois, de sorte qu'elles peuvent être appelées à tout moment.

Ajouté dans la version 3.8.

class unittest.**IsolatedAsyncioTestCase**(*methodName='runTest'*)

Cette classe fournit une API similaire à `TestCase` et accepte aussi les coroutines en tant que fonctions de test.

Ajouté dans la version 3.8.

loop_factory

The *loop_factory* passed to `asyncio.Runner`. Override in subclasses with `asyncio.EventLoop` to avoid using the asyncio policy system.

Ajouté dans la version 3.13.

coroutine asyncSetUp()

Méthode appelée pour réaliser la mise en place du test. Celle-ci est exécutée après `setUp()`. Elle est exécutée immédiatement avant l'appel de la méthode de test ; à l'exception de `AssertionError` ou

SkipTest, toute exception levée par cette méthode est considérée comme une erreur et non pas comme un échec du test. L'implémentation par défaut ne fait rien.

coroutine asyncTearDown()

Méthode appelée immédiatement après l'appel de la méthode de test et l'enregistrement du résultat. Elle est appelée avant `tearDown()`. Elle est appelée même si la méthode de test a levé une exception. De fait, l'implémentation d'une sous-classe doit être faite avec précaution si vous vérifiez l'état interne de la classe. Toute exception, autre que *AssertionError* ou *SkipTest*, levée par cette méthode est considérée comme une erreur supplémentaire plutôt que comme un échec du test (augmentant ainsi le nombre total des erreurs signalées). Cette méthode est appelée uniquement si l'exécution de *asyncSetUp()* est réussie quel que soit le résultat de la méthode de test. L'implémentation par défaut ne fait rien.

addAsyncCleanup(*function*, /, **args*, ***kwargs*)

Cette méthode accepte une coroutine qui peut être utilisée comme fonction de nettoyage.

coroutine enterAsyncContext(*cm*)

Enter the supplied *asynchronous context manager*. If successful, also add its `__aexit__()` method as a cleanup function by *addAsyncCleanup()* and return the result of the `__aenter__()` method.

Ajouté dans la version 3.11.

run(*result=None*)

Configure une nouvelle boucle d'évènements pour exécuter le test, en collectant le résultat dans l'objet *TestResult* passé comme *result*. Si *result* est omis ou vaut `None`, un objet temporaire de résultat est créé (en appelant la méthode `defaultTestResult()`) et utilisé. L'objet résultat est renvoyé à l'appelant de *run()*. À la fin du test, toutes les taches de la boucle d'événements sont annulées.

Exemple illustrant l'ordre :

```
from unittest import IsolatedAsyncioTestCase

events = []


class Test(IsolatedAsyncioTestCase):


    def setUp(self):
        events.append("setUp")

    async def asyncSetUp(self):
        self._async_connection = await AsyncConnection()
        events.append("asyncSetUp")

    async def test_response(self):
        events.append("test_response")
        response = await self._async_connection.get("https://example.com")
        self.assertEqual(response.status_code, 200)
        self.addAsyncCleanup(self.on_cleanup)

    def tearDown(self):
        events.append("tearDown")

    async def asyncTearDown(self):
        await self._async_connection.close()
        events.append("asyncTearDown")

    async def on_cleanup(self):
        events.append("cleanup")
```

(suite sur la page suivante)

(suite de la page précédente)

```
if __name__ == "__main__":
    unittest.main()
```

Après avoir lancé les tests, `events` contiendrait `["setUp", "asyncSetUp", "test_response", "asyncTearDown", "tearDown", "cleanup"]`.

class unittest.**FunctionTestCase**(*testFunc*, *setUp=None*, *tearDown=None*, *description=None*)

Cette classe implémente la partie de l'interface `TestCase` qui permet au lanceur de test de piloter le scénario de test, mais ne fournit pas les méthodes que le code test peut utiliser pour vérifier et signaler les erreurs. Ceci est utilisé pour créer des scénario de test utilisant du code de test existant afin de faciliter l'intégration dans un *framework* de test basé sur `unittest`.

Regroupement des tests

class unittest.**TestSuite**(*tests=()*)

Cette classe représente une agrégation de cas de test individuels et de suites de tests. La classe présente l'interface requise par le lanceur de test pour être exécutée comme tout autre cas de test. L'exécution d'une instance de `TestSuite` est identique à l'itération sur la suite, en exécutant chaque test indépendamment.

Si *tests* est fourni, il doit s'agir d'un itérable de cas de test individuels ou d'autres suites de test qui seront utilisés pour construire la suite initial. Des méthodes supplémentaires sont fournies pour ajouter ultérieurement des cas de test et des suites à la collection.

Les objets `TestSuite` se comportent comme les objets `TestCase`, sauf qu'ils n'implémentent pas réellement un test. Au lieu de cela, ils sont utilisés pour regrouper les tests en groupes de tests qui doivent être exécutés ensemble. Des méthodes supplémentaires sont disponibles pour ajouter des tests aux instances de `TestSuite` :

addTest(*test*)

Ajouter un objet `TestCase` ou `TestSuite` à la suite de tests.

addTests(*tests*)

Ajouter tous les tests d'un itérable d'instances de `TestCase` et de `TestSuite` à cette suite de tests.

C'est l'équivalent d'une itération sur *tests*, appelant `addTest()` pour chaque élément.

`TestSuite` partage les méthodes suivantes avec `TestCase` :

run(*result*)

Exécute les tests associés à cette suite, en collectant le résultat dans l'objet de résultat de test passé par *result*. Remarquer que contrairement à `TestCase.run()`, `TestSuite.run()` nécessite que l'objet résultat soit passé.

debug()

Exécute les tests associés à cette suite sans collecter le résultat. Ceci permet aux exceptions levées par le test d'être propagées à l'appelant et peut être utilisé pour exécuter des tests sous un débogueur.

countTestCases()

Renvoie le nombre de tests représentés par cet objet de test, y compris tous les tests individuels et les sous-suites.

__iter__()

Tests grouped by a `TestSuite` are always accessed by iteration. Subclasses can lazily provide tests by overriding `__iter__()`. Note that this method may be called several times on a single suite (for example when counting tests or comparing for equality) so the tests returned by repeated iterations before `TestSuite.run()` must be the same for each call iteration. After `TestSuite.run()`, callers should not rely on the tests returned by this method unless the caller uses a subclass that overrides `TestSuite._removeTestAtIndex()` to preserve test references.

Modifié dans la version 3.2 : In earlier versions the `TestSuite` accessed tests directly rather than through iteration, so overriding `__iter__()` wasn't sufficient for providing tests.

Modifié dans la version 3.4 : Dans les versions précédentes, la classe `TestSuite` contenait des références à chaque `TestCase` après l'appel à `TestSuite.run()`. Les sous-classes peuvent restaurer ce comportement en surchargeant `TestSuite._removeTestAtIndex()`.

Dans l'utilisation typique de l'objet `TestSuite`, la méthode `run()` est invoquée par une classe `TestRunner` plutôt que par le système de test de l'utilisateur.

Chargement et exécution des tests

class unittest.**TestLoader**

La classe `TestLoader` est utilisée pour créer des suites de tests à partir de classes et de modules. Normalement, il n'est pas nécessaire de créer une instance de cette classe ; le module `unittest` fournit une instance qui peut être partagée comme `unittest.defaultTestLoader`. L'utilisation d'une sous-classe ou d'une instance permet cependant de personnaliser certaines propriétés configurables.

Les objets de la classe `TestLoader` ont les attributs suivants :

errors

A list of the non-fatal errors encountered while loading tests. Not reset by the loader at any point. Fatal errors are signalled by the relevant method raising an exception to the caller. Non-fatal errors are also indicated by a synthetic test that will raise the original error when run.

Ajouté dans la version 3.5.

Les objets de la classe `TestLoader` ont les attributs suivants :

loadTestsFromTestCase(*testCaseClass*)

Renvoie une suite de tous les cas de test contenus dans la classe `TestCaseClass`dérivée de `testCase`.

Une instance de cas de test est créée pour chaque méthode nommée par `getTestCaseNames()`. Par défaut, ce sont les noms des méthodes commençant par "test". Si `getTestTestCaseNames()` ne renvoie aucune méthode, mais que la méthode `runTest()` est implémentée, un seul cas de test est créé pour cette méthode à la place.

loadTestsFromModule(*module*, *, *pattern=None*)

Renvoie une suite de tous les cas de test contenus dans le module donné. Cette méthode recherche *module* pour les classes dérivées de `TestCase` et crée une instance de la classe pour chaque méthode de test définie pour cette classe.

> **Note**
>
> Bien que l'utilisation d'une hiérarchie de classes `TestCase` (les classes dérivées de *TestCase*) peut être un moyen pratique de partager des *fixtures* et des fonctions utilitaires, définir une méthode de test pour des classes de base non destinées à être directement instanciée ne marche pas bien avec cette méthode. Cela peut toutefois s'avérer utile lorsque les *fixtures* sont différentes et définies dans des sous-classes.

Si un module fournit une fonction `load_tests`, il est appelé pour charger les tests. Cela permet aux modules de personnaliser le chargement des tests. C'est le protocole *load_tests protocol*. L'argument *pattern* est passé comme troisième argument à `load_tests`.

Modifié dans la version 3.2 : Ajout de la prise en charge de `load_tests`.

Modifié dans la version 3.5 : Support for a keyword-only argument *pattern* has been added.

Modifié dans la version 3.12 : The undocumented and unofficial *use_load_tests* parameter has been removed.

loadTestsFromName(*name*, *module=None*)

Renvoie une suite de tous les cas de test en fonction d'un spécificateur de chaîne de caractères.

Le spécificateur *name* est un "nom pointillé" qui peut être résolu soit par un module, une classe de cas de test, une méthode de test dans une classe de cas de test, une instance de `TestSuite`, ou un objet

appelable qui retourne une instance de classe `TestCase` ou de classe `TestSuite`. Ces contrôles sont appliqués dans l'ordre indiqué ici, c'est-à-dire qu'une méthode sur une classe de cas de test possible sera choisie comme "méthode de test dans une classe de cas de test", plutôt que comme "un objet appelable".

Par exemple, si vous avez un module `SampleTests` contenant une classe `TestCase` (classe dérivée de la classe `SampleTestCase`) avec trois méthodes de test (`test_one()`, `test_two()` et `test_three()`), l'élément spécificateur `SampleTests.sampleTestCase` renvoie une suite qui va exécuter les trois méthodes de tests. L'utilisation du spécificateur `SampleTests.SampleTestCase.test_two` permettrait de retourner une suite de tests qui ne lancerait que la méthode test `test_two()`. Le spécificateur peut se référer à des modules et packages qui n'ont pas été importés. Ils seront importés par un effet de bord.

La méthode résout facultativement *name* relatif au *module* donné.

Modifié dans la version 3.5 : Si une `ImportError` ou `AttributeError` se produit pendant la traversée de *name*, un test synthétique qui enrichie l'erreur produite lors de l'exécution est renvoyé. Ces erreurs sont incluses dans les erreurs accumulées par *self.errors*.

loadTestsFromNames(*names*, *module=None*)

Similaire à `loadTestsFromName()`, mais prend une séquence de noms plutôt qu'un seul nom. La valeur renvoyée est une suite de tests qui gère tous les tests définis pour chaque nom.

getTestCaseNames(*testCaseClass*)

Renvoie une séquence triée de noms de méthodes trouvés dans *testCaseClass* ; ceci doit être une sous-classe de `TestCase`.

discover(*start_dir*, *pattern='test*.py'*, *top_level_dir=None*)

Trouve tous les modules de test en parcourant les sous-répertoires du répertoire de démarrage spécifié, et renvoie un objet *TestSuite* qui les contient. Seuls les fichiers de test qui correspondent à *pattern* sont chargés. Seuls les noms de modules qui sont importables (c'est-à-dire qui sont des identifiants Python valides) sont chargés.

All test modules must be importable from the top level of the project. If the start directory is not the top level directory then *top_level_dir* must be specified separately.

Si l'importation d'un module échoue, par exemple en raison d'une erreur de syntaxe, celle-ci est alors enregistrée comme une erreur unique et la découverte se poursuit. Si l'échec de l'importation est dû au fait que `SkipTest` est levé, il est enregistré comme un saut plutôt que comme un message d'erreur.

Si un paquet (un répertoire contenant un fichier nommé `__init__.py`) est trouvé, le paquet est alors vérifié pour une fonction `load_tests`. Si elle existe, elle s'appellera `package.load_tests(loader, tests, pattern)`. Le mécanisme de découverte de test prend soin de s'assurer qu'un paquet n'est vérifié qu'une seule fois au cours d'une invocation, même si la fonction *load_tests* appelle elle-même `loader.discover`.

Si `load_tests` existe alors la découverte ne poursuit pas la récursion dans le paquet, `load_tests` a la responsabilité de charger tous les tests dans le paquet.

The pattern is deliberately not stored as a loader attribute so that packages can continue discovery themselves.

top_level_dir is stored internally, and used as a default to any nested calls to `discover()`. That is, if a package's `load_tests` calls `loader.discover()`, it does not need to pass this argument.

start_dir peut être un nom de module ainsi qu'un répertoire.

Ajouté dans la version 3.2.

Modifié dans la version 3.4 : Les modules levant une exception `SkipTest` au moment de leur importation ne sont pas considérés comme des erreurs, et sont marqués à sauter.

start_dir peut être un *paquet-espace de nommage*.

Les chemins sont classés avant d'être importés afin que l'ordre d'exécution soit toujours le même, même si le système de fichiers sous-jacent ne classe pas les fichiers par leur nom.

Modifié dans la version 3.5 : Les paquets trouvés sont maintenant vérifiés pour `load_tests` indépendamment du fait que leur chemin d'accès corresponde ou non à *pattern*, car il est impossible pour un nom

de paquet de correspondre au motif par défaut.

Modifié dans la version 3.11 : *start_dir* can not be a *namespace packages*. It has been broken since Python 3.7, and Python 3.11 officially removes it.

Modifié dans la version 3.13 : *top_level_dir* is only stored for the duration of *discover* call.

Modifié dans la version 3.14 : *start_dir* can once again be a *namespace package*.

Les attributs suivants d'une classe `TestLoader` peuvent être configurés soit par héritage, soit par affectation sur une instance :

testMethodPrefix

Chaîne donnant le préfixe des noms de méthodes qui seront interprétés comme méthodes de test. La valeur par défaut est `'test'`.

This affects `getTestCaseNames()` and all the `loadTestsFrom*` methods.

sortTestMethodsUsing

Function to be used to compare method names when sorting them in `getTestCaseNames()` and all the `loadTestsFrom*` methods.

suiteClass

Objet appelable qui construit une suite de tests à partir d'une liste de tests. Aucune méthode sur l'objet résultant n'est nécessaire. La valeur par défaut est la classe `TestSuite`.

This affects all the `loadTestsFrom*` methods.

testNamePatterns

List of Unix shell-style wildcard test name patterns that test methods have to match to be included in test suites (see `-k` option).

If this attribute is not `None` (the default), all test methods to be included in test suites must match one of the patterns in this list. Note that matches are always performed using `fnmatch.fnmatchcase()`, so unlike patterns passed to the `-k` option, simple substring patterns will have to be converted using `*` wildcards.

This affects all the `loadTestsFrom*` methods.

Ajouté dans la version 3.7.

class unittest.**TestResult**

Cette classe est utilisée pour compiler des informations sur les tests qui ont réussi et ceux qui ont échoué.

Un objet `TestResult` stocke les résultats d'un ensemble de tests. Les classes `TestCase` et `TestSuite` s'assurent que les résultats sont correctement enregistrés. Les auteurs du test n'ont pas à se soucier de l'enregistrement des résultats des tests.

Les cadriciels de test construits sur `unittest` peuvent nécessiter l'accès à l'objet `TestResult` généré en exécutant un ensemble de tests à des fins de génération de comptes-rendu. Une instance de `TestResult` est alors renvoyée par la méthode `TestRunner.run()` à cette fin.

Les instance de `TestResult` ont les attributs suivants qui sont intéressant pour l'inspection des résultats de l'exécution d'un ensemble de tests :

errors

Une liste contenant des paires d'instances de `TestCase` et de chaînes de caractères contenant des traces de pile d'appels formatées. Chaque paire représente un test qui a levé une exception inattendue.

failures

A list containing 2-tuples of `TestCase` instances and strings holding formatted tracebacks. Each tuple represents a test where a failure was explicitly signalled using the *assert* methods*.

skipped

Une liste contenant des paires d'instances de `TestCase` et de chaînes de caractères contenant la raison de l'omission du test.

Ajouté dans la version 3.1.

expectedFailures

Une liste contenant des paires d'instances `TestCase` et de chaînes de caractères contenant des traces de pile d'appels formatées. Chaque paire représente un échec attendu ou une erreur du scénario de test.

unexpectedSuccesses

Une liste contenant les instances `TestCase` qui ont été marquées comme des échecs attendus, mais qui ont réussi.

collectedDurations

A list containing 2-tuples of test case names and floats representing the elapsed time of each test which was run.

Ajouté dans la version 3.12.

shouldStop

A positionner sur `True` quand l'exécution des tests doit être arrêter par `stop()`.

testsRun

Le nombre total de tests effectués jusqu'à présent.

buffer

S'il est défini sur *true*, `sys.stdout` et `sys.stderr` sont mis dans un tampon entre les appels de `startTest()` et `stopTest()`. La sortie collectée est répercutée sur les sorties `sys.stdout` et `sys.stderr` réels uniquement en cas d'échec ou d'erreur du test. Toute sortie est également attachée au message d'erreur.

Ajouté dans la version 3.2.

failfast

Si la valeur est *true* `stop()` est appelée lors de la première défaillance ou erreur, ce qui interrompt le test en cours d'exécution.

Ajouté dans la version 3.2.

tb_locals

Si la valeur est *true*, les variables locales sont affichées dans les traces d'appels.

Ajouté dans la version 3.5.

wasSuccessful()

Renvoie `True` si tous les tests effectués jusqu'à présent ont réussi, sinon renvoie `False`.

Modifié dans la version 3.4 : Renvoie `False` s'il y a eu des `unexpectedSuccesses` dans les tests annotés avec le décorateur `expectedFailure()`.

stop()

Cette méthode peut être appelée pour signaler que l'ensemble des tests en cours d'exécution doit être annulé en définissant l'attribut `shouldStop` sur `True`. Les instances de `TestRunner` doivent respecter ce signal et se terminer sans exécuter de tests supplémentaires.

Par exemple, cette fonctionnalité est utilisée par la classe `TextTestRunner` pour arrêter le cadriciel de test lorsque l'utilisateur lance une interruption clavier. Les outils interactifs qui fournissent des implémentations de `TestRunner` peuvent l'utiliser de la même manière.

Les méthodes suivantes de la classe `TestResult` sont utilisées pour maintenir les structures de données internes, et peuvent être étendues dans des sous-classes pour gérer des exigences supplémentaires en termes de compte-rendu. Cette fonction est particulièrement utile pour créer des outils qui prennent en charge la génération de rapports interactifs pendant l'exécution des tests.

startTest(*test*)

Appelé lorsque le scénario de test *test* est sur le point d'être exécuté.

stopTest(*test*)

Appelé après l'exécution du cas de test *test*, quel qu'en soit le résultat.

startTestRun()

Appelé une fois avant l'exécution des tests.

Ajouté dans la version 3.1.

stopTestRun()

Appelé une fois après l'exécution des tests.

Ajouté dans la version 3.1.

addError(*test*, *err*)

Appelé lorsque le cas de test *test* soulève une exception inattendue. *err* est un *n*-uplet de la même forme que celle renvoyée par `sys.exc_info()` : `(type, valeur, traceback)`.

L'implémentation par défaut ajoute un *n*-uplet `(test, formatted_err)` à l'attribut `errors` de l'instance, où *formatted_err* est une trace formatée à partir de *err*.

addFailure(*test*, *err*)

Appelé lorsque le cas de test *test* soulève une exception inattendue. *err* est un triplet de la même forme que celui renvoyé par `sys.exc_info()` : `(type, valeur, traceback)`.

L'implémentation par défaut ajoute un *n*-uplet `(test, formatted_err)` à l'attribut `errors` de l'instance, où *formatted_err* est une trace formatée à partir de *err*.

addSuccess(*test*)

Appelé lorsque le scénario de test *test* réussit.

L'implémentation par défaut ne fait rien.

addSkip(*test*, *reason*)

Appelé lorsque le scénario de test *test* est ignoré. *raison* est la raison pour laquelle le test donné à été ignoré.

L'implémentation par défaut ajoute un *n*-uplet `(test, raison)` à l'attribut `skipped` de l'instance.

addExpectedFailure(*test*, *err*)

Appelé lorsque le scénario de test *test* échoue (assertion ou exception), mais qui a été marqué avec le décorateur `expectedFailure()`.

L'implémentation par défaut ajoute un *n*-uplet `(test, formatted_err)` à l'attribut `errors` de l'instance, où *formatted_err* est une trace formatée à partir de *err*.

addUnexpectedSuccess(*test*)

Appelé lorsque le scénario de test *test* réussit, mais que ce scénario a été marqué avec le décorateur `expectedFailure()`.

L'implémentation par défaut ajoute le test à l'attribut `unexpectedSuccesses` de l'instance.

addSubTest(*test*, *subtest*, *outcome*)

Appelé à la fin d'un sous-test. *test* est le cas de test correspondant à la méthode de test. *subtest* est une instance dérivée de `TestCase` décrivant le sous-test.

Si *outcome* est `None`, le sous-test a réussi. Sinon, il a échoué avec une exception où *outcome* est un triplet du formulaire renvoyé par `sys.exc_info()` : `(type, valeur, traceback)`.

L'implémentation par défaut ne fait rien lorsque le résultat est un succès, et enregistre les échecs de sous-test comme des échecs normaux.

Ajouté dans la version 3.4.

addDuration(*test*, *elapsed*)

Called when the test case finishes. *elapsed* is the time represented in seconds, and it includes the execution of cleanup functions.

Ajouté dans la version 3.12.

class unittest.**TextTestResult**(*stream*, *descriptions*, *verbosity*, *, *durations=None*)

A concrete implementation of `TestResult` used by the `TextTestRunner`. Subclasses should accept `**kwargs` to ensure compatibility as the interface changes.

Ajouté dans la version 3.2.

Modifié dans la version 3.12 : Added the *durations* keyword parameter.

unittest.**defaultTestLoader**

Instance de la classe `TestLoader` destinée à être partagée. Si aucune personnalisation de la classe `TestLoader` n'est nécessaire, cette instance peut être utilisée au lieu de créer plusieurs fois de nouvelles instances.

class unittest.**TextTestRunner**(*stream=None*, *descriptions=True*, *verbosity=1*, *failfast=False*, *buffer=False*, *resultclass=None*, *warnings=None*, *, *tb_locals=False*, *durations=None*)

Une implémentation de base d'un lanceur de test qui fournit les résultats dans un flux. Si *stream* est `None`, par défaut, `sys.stderr` est utilisé comme flux de sortie. Cette classe a quelques paramètres configurables, mais est essentiellement très simple. Les applications graphiques qui exécutent des suites de tests doivent fournir des implémentations alternatives. De telles implémentations doivent accepter `**kwargs` car l'interface pour construire les lanceurs change lorsque des fonctionnalités sont ajoutées à *unittest*.

By default this runner shows `DeprecationWarning`, `PendingDeprecationWarning`, `ResourceWarning` and `ImportWarning` even if they are *ignored by default*. This behavior can be overridden using Python's `-Wd` or `-Wa` options (see Warning control) and leaving *warnings* to `None`.

Modifié dans la version 3.2 : Added the *warnings* parameter.

Modifié dans la version 3.2 : Le flux par défaut est défini sur `sys.stderr` au moment de l'instanciation plutôt qu'à l'importation.

Modifié dans la version 3.5 : Added the *tb_locals* parameter.

Modifié dans la version 3.12 : Added the *durations* parameter.

_makeResult()

Cette méthode renvoie l'instance de `TestResult` utilisée par `run()`. Il n'est pas destiné à être appelé directement, mais peut être surchargée dans des sous-classes pour fournir un `TestResult` personnalisé.

`_makeResult()` instancie la classe ou l'appelable passé dans le constructeur `TextTestRunner` comme argument `resultclass`. Il vaut par défaut `TextTestResult` si aucune `resultclass` n'est fournie. La classe de résultat est instanciée avec les arguments suivants :

```
stream, descriptions, verbosity
```

run(*test*)

Cette méthode est l'interface publique principale du `TextTestRunner`. Cette méthode prend une instance `TestSuite` ou `TestCase`. Un `TestResult` est créé en appelant `_makeResult()` et le ou les tests sont exécutés et les résultats affichés sur la sortie standard.

unittest.**main**(*module='__main__'*, *defaultTest=None*, *argv=None*, *testRunner=None*, *testLoader=unittest.defaultTestLoader*, *exit=True*, *verbosity=1*, *failfast=None*, *catchbreak=None*, *buffer=None*, *warnings=None*)

Un programme en ligne de commande qui charge un ensemble de tests à partir du *module* et les exécute. L'utilisation principale est de rendre les modules de test facilement exécutables. L'utilisation la plus simple pour cette fonction est d'inclure la ligne suivante à la fin d'un script de test :

```
if __name__ == '__main__':
    unittest.main()
```

Vous pouvez exécuter des tests avec des informations plus détaillées en utilisant l'option de verbosité :

```
if __name__ == '__main__':
    unittest.main(verbosity=2)
```

L'argument *defaultTest* est soit le nom d'un seul test, soit un itérable de noms de test à exécuter si aucun nom de test n'est spécifié via *argv*. Si aucun nom de test n'est fourni via *argv*, tous les tests trouvés dans *module* sont exécutés.

L'argument *argv* peut être une liste d'options passées au programme, le premier élément étant le nom du programme. S'il n'est pas spécifié ou vaut `None`, les valeurs de `sys.argv` sont utilisées.

The *testRunner* argument can either be a test runner class or an already created instance of it. By default `main` calls `sys.exit()` with an exit code indicating success (0) or failure (1) of the tests run. An exit code of 5 indicates that no tests were run or skipped.

L'argument *testLoader* doit être une instance de `TestLoader`, et par défaut de `defaultTestLoader`.

Les `main` sont utilisés à partir de l'interpréteur interactif en passant dans l'argument `exit=False`. Ceci affiche le résultat sur la sortie standard sans appeler `sys.exit()` :

```
>>> from unittest import main
>>> main(module='test_module', exit=False)
```

Les paramètres *failfast*, *catchbreak* et *buffer* ont le même effet que la même option en ligne de commande *command-line options*.

L'argument *warnings* spécifie l'argument *filtre d'avertissement* qui doit être utilisé lors de l'exécution des tests. Si elle n'est pas spécifiée, elle reste réglée sur `None` si une option `-W'` est passée à **python** (voir Utilisation des avertissements), sinon elle sera réglée sur `'default'`.

Calling `main` returns an object with the `result` attribute that contains the result of the tests run as a `unittest.TestResult`.

Modifié dans la version 3.1 : Ajout du paramètre *exit*.

Modifié dans la version 3.2 : Ajout des paramètres *verbosity*, *failfast*, *catchbreak*, *buffer* et *warnings*.

Modifié dans la version 3.4 : Le paramètre *defaultTest* a été modifié pour accepter également un itérable de noms de test.

Protocole de chargement des tests (*load_tests Protocol*)

Ajouté dans la version 3.2.

Les modules ou paquets peuvent personnaliser la façon dont les tests sont chargés à partir de ceux-ci pendant l'exécution des tests ou pendant la découverte de tests en implémentant une fonction appelée `load_tests`.

Si un module de test définit `load_tests` il est appelé par `TestLoader.loadTestsFromModule()` avec les arguments suivants :

```
load_tests(loader, standard_tests, pattern)
```

où *pattern* est passé directement depuis `loadTestsFromModule`. La valeur par défaut est `None`.

Elle doit renvoyer une classe `TestSuite`.

loader est l'instance de `TestLoader` qui effectue le chargement. *standard_tests* sont les tests qui sont chargés par défaut depuis le module. Il est courant que les modules de test veuillent seulement ajouter ou supprimer des tests de l'ensemble standard de tests. Le troisième argument est utilisé lors du chargement de paquets dans le cadre de la découverte de tests.

Une fonction typique de `load_tests` qui charge les tests d'un ensemble spécifique de classes `TestCase` peut ressembler à :

```
test_cases = (TestCase1, TestCase2, TestCase3)

def load_tests(loader, tests, pattern):
    suite = TestSuite()
    for test_class in test_cases:
```

(suite sur la page suivante)

(suite de la page précédente)

```
        tests = loader.loadTestsFromTestCase(test_class)
        suite.addTests(tests)
    return suite
```

Si la découverte est lancée dans un répertoire contenant un paquet, soit à partir de la ligne de commande, soit en appelant *TestLoader.discover()*, alors le système recherche dans le fichier du paquet `__init__.py` la fonction `load_tests`. Si cette fonction n'existe pas, la découverte se poursuit dans le paquet comme si c'était juste un autre répertoire. Sinon, la découverte des tests du paquet est effectuée par `load_tests` qui est appelé avec les arguments suivants :

```
load_tests(loader, standard_tests, pattern)
```

Doit renvoyer une classe *TestSuite* représentant tous les tests du paquet. (`standard_tests` ne contient que les tests collectés dans le fichier `__init__.py`).

Comme le motif est passé à `load_tests`, le paquet est libre de continuer (et potentiellement de modifier) la découverte des tests. Une fonction « ne rien faire » `load_tests` pour un paquet de test ressemblerait à :

```
def load_tests(loader, standard_tests, pattern):
    # top level directory cached on loader instance
    this_dir = os.path.dirname(__file__)
    package_tests = loader.discover(start_dir=this_dir, pattern=pattern)
    standard_tests.addTests(package_tests)
    return standard_tests
```

Modifié dans la version 3.5 : La découverte de test ne vérifie plus que les noms de paquets correspondent à *pattern* en raison de l'impossibilité de trouver des noms de paquets correspondant au motif par défaut.

27.5.9 Classes et modules d'aménagements des tests

Les classes et modules d'aménagements des tests sont implémentés dans *TestSuite*. Lorsque la suite de tests rencontre un test d'une nouvelle classe, alors `tearDownClass()` de la classe précédente (s'il y en a une) est appelé, suivi de `setUpClass()` de la nouvelle classe.

De même, si un test provient d'un module différent du test précédent, alors `tearDownModule` du module précédent est exécuté, suivi par `setUpModule` du nouveau module.

Après que tous les tests ont été exécutés, les `tearDownClass` et `tearDownModule` finaux sont exécutés.

Notez que les aménagements de tests partagés ne fonctionnent pas bien avec de « potentielles » fonctions comme la parallélisation de test et qu'ils brisent l'isolation des tests. Ils doivent être utilisés avec parcimonie.

L'ordre par défaut des tests créés par les chargeurs de tests unitaires est de regrouper tous les tests des mêmes modules et classes. Cela à pour conséquence que `setUpClass` / `setUpModule` (etc) sont appelé exactement une fois par classe et module. Si vous rendez l'ordre aléatoire, de sorte que les tests de différents modules et classes soient adjacents les uns aux autres, alors ces fonctions d'aménagements partagées peuvent être appelées plusieurs fois dans un même test.

Les aménagements de tests partagés ne sont pas conçus pour fonctionner avec des suites dont la commande n'est pas standard. Une `BaseTestSuite` existe toujours pour les cadriciels qui ne veulent pas gérer les aménagements de tests partagés.

S'il y a des exceptions levées pendant l'une des fonctions d'aménagement de tests partagés, le test est signalé comme étant en erreur. Parce qu'il n'y a pas d'instance de test correspondante, un objet `_ErrorHolder` (qui a la même interface qu'une classe *TestCase*) est créé pour représenter l'erreur. Si vous n'utilisez que le lanceur de test unitaire standard, ce détail n'a pas d'importance, mais si vous êtes un auteur de cadriciel de test, il peut être pertinent.

Classes de mise en place (*setUpClass*) et de démantèlement des tests (*tearDownClass*)

Elles doivent être implémentées en tant que méthodes de classe :

```
import unittest

class Test(unittest.TestCase):
    @classmethod
    def setUpClass(cls):
        cls._connection = createExpensiveConnectionObject()

    @classmethod
    def tearDownClass(cls):
        cls._connection.destroy()
```

Si vous voulez que les classes de base `setUpClass` et `tearDownClass` soient appelées, vous devez les appeler vous-même. Les implémentations dans `TestCase` sont vides.

Si une exception est levée pendant l'exécution de `setUpClass` alors les tests dans la classe ne sont pas exécutés et la classe `tearDownClass` n'est pas exécutée. Les classes ignorées n'auront pas d'exécution de `setUpClass` ou `tearDownClass`. Si l'exception est une exception `SkipTest` alors la classe est signalée comme ayant été ignorée au lieu d'être en échec.

Module de mise en place (*setUpModule*) et de démantèlement des tests (*tearDownModule*)

Elles doivent être implémentées en tant que fonctions :

```
def setUpModule():
    createConnection()

def tearDownModule():
    closeConnection()
```

Si une exception est levée pendant l'exécution de la fonction `setUpModule` alors aucun des tests du module ne sera exécuté et la fonction `tearDownModule` ne sera pas exécutée. Si l'exception est une exception `SkipTest` alors le module est signalé comme ayant été ignoré au lieu d'être en échec.

Pour ajouter du code de nettoyage qui doit être exécuté même en cas d'exception, utilisez `addModuleCleanup` :

`unittest.`**`addModuleCleanup`**(*function*, */*, **args*, ***kwargs*)

Ajout d'une fonction à appeler après `tearDownModule()` pour nettoyer les ressources utilisées pendant le test. Les fonctions seront appelées dans l'ordre inverse de l'ordre dans lequel elles ont été ajoutées (LIFO). Elles sont appelées avec tous les arguments et arguments nommés passés à `addModuleCleanup()` quand elles sont ajoutées.

Si `setUpModule()` échoue, impliquant que `tearDownModule()` n'est pas appelée, alors les fonctions de nettoyage ajoutées sont quand même toujours appelées.

Ajouté dans la version 3.8.

classmethod `unittest.`**`enterModuleContext`**(*cm*)

Enter the supplied *context manager*. If successful, also add its `__exit__()` method as a cleanup function by `addModuleCleanup()` and return the result of the `__enter__()` method.

Ajouté dans la version 3.11.

`unittest.`**`doModuleCleanups`**()

Cette méthode est appelée sans conditions après `tearDownModule()`, ou après `setUpModule()` si `setUpModule()` lève une exception.

It is responsible for calling all the cleanup functions added by `addModuleCleanup()`. If you need cleanup functions to be called *prior* to `tearDownModule()` then you can call `doModuleCleanups()` yourself.

`doModuleCleanups()` extrait les méthodes de la pile des fonctions de nettoyage une à la fois, de sorte qu'elles peuvent être appelées à tout moment.

Ajouté dans la version 3.8.

27.5.10 Traitement des signaux

Ajouté dans la version 3.2.

The `-c/--catch` command-line option to unittest, along with the `catchbreak` parameter to `unittest.main()`, provide more friendly handling of control-C during a test run. With catch break behavior enabled control-C will allow the currently running test to complete, and the test run will then end and report all the results so far. A second control-c will raise a `KeyboardInterrupt` in the usual way.

Le gestionnaire du signal *contrôle-C* tente de rester compatible avec le code ou les tests qui installent leur propre gestionnaire `signal.SIGINT`. Si le gestionnaire `unittest` est appelé mais *n'est pas* le gestionnaire installé `signal.SIGINT`, c'est-à-dire qu'il a été remplacé par le système sous test et délégué, alors il appelle le gestionnaire par défaut. C'est normalement le comportement attendu par un code qui remplace un gestionnaire installé et lui délègue. Pour les tests individuels qui ont besoin que le signal *contrôle-C* "*unittest*" soit désactivée, le décorateur `removeHandler()` peut être utilisé.

Il existe quelques fonctions de support pour les auteurs de cadriciel afin d'activer la fonctionnalité de gestion des *contrôle-C* dans les cadriciels de test.

`unittest.`**`installHandler`**`()`

Installe le gestionnaire *contrôle-c*. Quand un `signal.SIGINT` est reçu (généralement en réponse à l'utilisateur appuyant sur *contrôle-c*) tous les résultats enregistrés vont appeler la méthode `stop()`.

`unittest.`**`registerResult`**`(`*result*`)`

Enregistre un objet `TestResult` pour la gestion du *contrôle-C*. L'enregistrement d'un résultat stocke une référence faible sur celui-ci, de sorte qu'il n'empêche pas que le résultat soit collecté par le ramasse-miette.

L'enregistrement d'un objet `TestResult` n'a pas d'effets de bord si la gestion du *contrôle-c* n'est pas activée, donc les cadriciels de test peuvent enregistrer sans condition tous les résultats qu'ils créent indépendamment du fait que la gestion soit activée ou non.

`unittest.`**`removeResult`**`(`*result*`)`

Supprime un résultat enregistré. Une fois qu'un résultat a été supprimé, `stop()` n'est plus appelé sur cet objet résultat en réponse à un *contrôle-c*.

`unittest.`**`removeHandler`**`(`*function=None*`)`

Lorsqu'elle est appelée sans arguments, cette fonction supprime le gestionnaire *contrôle-c* s'il a été installé. Cette fonction peut également être utilisée comme décorateur de test pour supprimer temporairement le gestionnaire pendant l'exécution du test :

```
@unittest.removeHandler
def test_signal_handling(self):
    ...
```

27.6 `unittest.mock` --- mock object library

Ajouté dans la version 3.3.

Code source : Lib/unittest/mock.py

`unittest.mock` est une bibliothèque pour tester en Python. Elle permet de remplacer des parties du système sous tests par des objets simulacres et faire des assertions sur la façon dont ces objets ont été utilisés.

`unittest.mock` fournit une classe `Mock` pour ne pas avoir besoin de créer manuellement des objets factices dans la suite de tests. Après avoir effectué une action, on peut faire des assertions sur les méthodes / attributs utilisés et les

arguments avec lesquels ils ont été appelés. On peut également spécifier des valeurs renvoyées et définir les attributs nécessaires aux tests.

De plus, *mock* fournit un décorateur `patch()` qui est capable de *patcher* les modules et les classes dans la portée d'un test, ainsi que `sentinel` pour créer des objets uniques. Voir le guide rapide *quick guide* pour quelques exemples d'utilisation de `Mock`, `MagicMock` et `patch()`.

Mock is designed for use with `unittest` and is based on the 'action -> assertion' pattern instead of 'record -> replay' used by many mocking frameworks.

There is a backport of `unittest.mock` for earlier versions of Python, available as mock on PyPI.

27.6.1 Guide rapide

Les classes `Mock` et `MagicMock` créent tous les attributs et méthodes au fur et à mesure des accès et stockent les détails de la façon dont ils ont été utilisés. On peut les configurer, pour spécifier des valeurs de renvoi ou limiter les attributs utilisables, puis faire des assertions sur la façon dont ils ont été utilisés :

```
>>> from unittest.mock import MagicMock
>>> thing = ProductionClass()
>>> thing.method = MagicMock(return_value=3)
>>> thing.method(3, 4, 5, key='value')
3
>>> thing.method.assert_called_with(3, 4, 5, key='value')
```

`side_effect` allows you to perform side effects, including raising an exception when a mock is called :

```
>>> from unittest.mock import Mock
>>> mock = Mock(side_effect=KeyError('foo'))
>>> mock()
Traceback (most recent call last):
 ...
KeyError: 'foo'
```

```
>>> values = {'a': 1, 'b': 2, 'c': 3}
>>> def side_effect(arg):
...     return values[arg]
...
>>> mock.side_effect = side_effect
>>> mock('a'), mock('b'), mock('c')
(1, 2, 3)
>>> mock.side_effect = [5, 4, 3, 2, 1]
>>> mock(), mock(), mock()
(5, 4, 3)
```

Il existe beaucoup d'autres façons de configurer et de contrôler le comportement de *Mock*. Par exemple, l'argument *spec* configure le *mock* pour qu'il utilise les spécifications d'un autre objet. Tenter d'accéder à des attributs ou méthodes sur le *mock* qui n'existent pas sur l'objet *spec* lève une `AttributeError`.

The `patch()` decorator / context manager makes it easy to mock classes or objects in a module under test. The object you specify will be replaced with a mock (or other object) during the test and restored when the test ends :

```
>>> from unittest.mock import patch
>>> @patch('module.ClassName2')
... @patch('module.ClassName1')
... def test(MockClass1, MockClass2):
...     module.ClassName1()
...     module.ClassName2()
...     assert MockClass1 is module.ClassName1
...     assert MockClass2 is module.ClassName2
```

(suite sur la page suivante)

(suite de la page précédente)

```
...     assert MockClass1.called
...     assert MockClass2.called
...
>>> test()
```

Note

Lorsque l'on imbrique des décorateurs de patchs, les *mocks* sont transmis à la fonction décorée dans le même ordre qu'ils ont été déclarés (l'ordre normal *Python* des décorateurs est appliqué). Cela signifie du bas vers le haut, donc dans l'exemple ci-dessus, l'objet simulacre pour `module.ClassName1` est passé en premier.

Avec `patch()`, il est important de *patcher* les objets dans l'espace de nommage où ils sont recherchés. C'est ce qui se fait normalement, mais pour un guide rapide, lisez *où patcher*.

Comme tout décorateur, `patch()` peut être utilisé comme gestionnaire de contexte avec une instruction *with* :

```
>>> with patch.object(ProductionClass, 'method', return_value=None) as mock_method:
...     thing = ProductionClass()
...     thing.method(1, 2, 3)
...
>>> mock_method.assert_called_once_with(1, 2, 3)
```

Il existe également `patch.dict()` pour définir des valeurs d'un dictionnaire au sein d'une portée et restaurer ce dictionnaire à son état d'origine lorsque le test se termine :

```
>>> foo = {'key': 'value'}
>>> original = foo.copy()
>>> with patch.dict(foo, {'newkey': 'newvalue'}, clear=True):
...     assert foo == {'newkey': 'newvalue'}
...
>>> assert foo == original
```

Mock gère le remplacement des *méthodes magiques* de Python. La façon la plus simple d'utiliser les méthodes magiques est la classe `MagicMock`. Elle permet de faire des choses comme :

```
>>> mock = MagicMock()
>>> mock.__str__.return_value = 'foobarbaz'
>>> str(mock)
'foobarbaz'
>>> mock.__str__.assert_called_with()
```

Mock permet d'assigner des fonctions (ou d'autres instances *Mock*) à des méthodes magiques et elles seront appelées correctement. La classe `MagicMock` est juste une variante de *Mock* qui a toutes les méthodes magiques pré-créées (enfin, toutes les méthodes utiles).

L'exemple suivant est un exemple de création de méthodes magiques avec la classe *Mock* ordinaire :

```
>>> mock = Mock()
>>> mock.__str__ = Mock(return_value='wheeeeee')
>>> str(mock)
'wheeeeee'
```

Pour être sûr que les objets simulacres dans vos tests ont la même API que les objets qu'ils remplacent, utilisez *l'auto-spécification*. L'auto-spécification peut se faire via l'argument *autospec* de patch ou par la fonction `create_autospec()`. L'auto-spécification crée des objets simulacres qui ont les mêmes attributs et méthodes que les objets qu'ils remplacent, et toutes les fonctions et méthodes (y compris les constructeurs) ont les mêmes signatures d'appel que l'objet réel.

Ceci garantit que vos objets simulacres échouent de la même manière que votre code de production s'ils ne sont pas utilisés correctement :

```
>>> from unittest.mock import create_autospec
>>> def function(a, b, c):
...     pass
...
>>> mock_function = create_autospec(function, return_value='fishy')
>>> mock_function(1, 2, 3)
'fishy'
>>> mock_function.assert_called_once_with(1, 2, 3)
>>> mock_function('wrong arguments')
Traceback (most recent call last):
 ...
TypeError: missing a required argument: 'b'
```

La fonction `create_autospec()` peut aussi être utilisée sur les classes, où elle copie la signature de la méthode `__init__`, et sur les objets appelables où elle copie la signature de la méthode `__call__`.

27.6.2 La classe *Mock*

La classe `Mock` est un objet simulacre flexible destiné à remplacer l'utilisation d'objets bouchons et factices dans votre code. Les Mocks sont appelables et créent des attributs comme de nouveaux *Mocks* lorsque l'on y accède [1]. L'accès au même attribut renvoie toujours le même *mock*. Les simulacres enregistrent la façon dont ils sont utilisés, ce qui permet de faire des assertions sur ce que le code leur a fait.

La classe `MagicMock` est une sous-classe de `Mock` avec toutes les méthodes magiques pré-créées et prête à l'emploi. Il existe également des variantes non appelables, utiles lorsque l'on simule des objets qui ne sont pas appelables : `NonCallableMock` et `NonCallableMagicMock`

Le décorateur `patch()` facilite le remplacement temporaire de classes d'un module avec un objet `Mock`. Par défaut `patch()` crée un `MagicMock`. On peut spécifier une classe alternative de `Mock` en utilisant le paramètre *new_callable* de `patch()`.

class unittest.mock.**Mock**(*spec=None*, *side_effect=None*, *return_value=DEFAULT*, *wraps=None*, *name=None*, *spec_set=None*, *unsafe=False*, ***kwargs*)

Crée un nouvel objet `Mock`. `Mock` prend plusieurs arguments optionnels qui spécifient le comportement de l'objet *Mock* :

— *spec* : une liste de chaînes de caractères ou un objet existant (une classe ou une instance) qui sert de spécification pour l'objet simulacre. Si on passe un objet, alors une liste de chaînes de caractères est formée en appelant la fonction *dir* sur l'objet (à l'exclusion des attributs et méthodes magiques non pris en charge). L'accès à un attribut qui n'est pas dans cette liste entraîne la levée d'une exception `AttributeError`.

If *spec* is an object (rather than a list of strings) then `__class__` returns the class of the spec object. This allows mocks to pass `isinstance()` tests.

— *spec_set* : variante plus stricte de *spec*. S'il est utilisé, essayer d'utiliser la fonction *set* ou tenter d'accéder à un attribut sur le *mock* qui n'est pas sur l'objet passé comme *spec_set* lève une exception `AttributeError`.

— *side_effect* : fonction à appeler à chaque fois que le *Mock* est appelé. Voir l'attribut `side_effect`. Utile pour lever des exceptions ou modifier dynamiquement les valeurs de retour. La fonction est appelée avec les mêmes arguments que la fonction simulée et, à moins qu'elle ne renvoie `DEFAULT`, la valeur de retour de cette fonction devient la valeur de retour de la fonction simulée.

side_effect peut être soit une classe, soit une instance d'exception. Dans ce cas, l'exception est levée lors de l'appel de l'objet simulacre.

Si *side_effect* est un itérable alors chaque appel au *mock* renvoie la valeur suivante de l'itérable.

Utilisez `None` pour remettre à zéro un *side_effect*.

1. The only exceptions are magic methods and attributes (those that have leading and trailing double underscores). Mock doesn't create these but instead raises an `AttributeError`. This is because the interpreter will often implicitly request these methods, and gets *very* confused to get a new Mock object when it expects a magic method. If you need magic method support see *magic methods*.

— *return_value* : valeur renvoyée lors de l'appel de l'objet simulacre. Par défaut, il s'agit d'un nouveau *Mock* (créé lors du premier accès). Voir l'attribut `return_value`.
— *unsafe* : By default, accessing any attribute whose name starts with *assert*, *assret*, *asert*, *aseert* or *assrt* will raise an `AttributeError`. Passing `unsafe=True` will allow access to these attributes.

  Ajouté dans la version 3.5.
— *wraps* : élément que le simulacre doit simuler. Si *wraps* n'est pas `None` alors appeler *Mock* passe l'appel à l'objet simulé (renvoyant le résultat réel). L'accès à un attribut sur le *mock* renvoie un objet *Mock* qui simule l'attribut correspondant de l'objet simulé (donc essayer d'accéder à un attribut qui n'existe pas lève une exception `AttributeError`).

  Si l'objet simulacre a un ensemble explicite de *return_value* alors les appels ne sont pas passés à l'objet simulé et c'est *return_value* qui est renvoyée à la place.
— *name* : Si le *mock* a un nom, il est alors utilisé par la fonction *repr* du *mock*. C'est utile pour le débogage. Le nom est propagé aux enfants de l'objet *mock*.

Les *mocks* peuvent aussi être appelés avec des arguments par mots-clés arbitraires. Ceux-ci sont utilisés pour définir les attributs sur le *mock* après sa création. Voir la méthode `configure_mock()` pour plus de détails.

assert_called()

Asserter que le *mock* a été appelé au moins une fois.

```
>>> mock = Mock()
>>> mock.method()
<Mock name='mock.method()' id='...'>
>>> mock.method.assert_called()
```

Ajouté dans la version 3.6.

assert_called_once()

Asserter que le *mock* a été appelé exactement une fois.

```
>>> mock = Mock()
>>> mock.method()
<Mock name='mock.method()' id='...'>
>>> mock.method.assert_called_once()
>>> mock.method()
<Mock name='mock.method()' id='...'>
>>> mock.method.assert_called_once()
Traceback (most recent call last):
...
AssertionError: Expected 'method' to have been called once. Called 2 times.
Calls: [call(), call()].
```

Ajouté dans la version 3.6.

assert_called_with(**args*, ***kwargs*)

This method is a convenient way of asserting that the last call has been made in a particular way :

```
>>> mock = Mock()
>>> mock.method(1, 2, 3, test='wow')
<Mock name='mock.method()' id='...'>
>>> mock.method.assert_called_with(1, 2, 3, test='wow')
```

assert_called_once_with(**args*, ***kwargs*)

Assert that the mock was called exactly once and that call was with the specified arguments.

```
>>> mock = Mock(return_value=None)
>>> mock('foo', bar='baz')
>>> mock.assert_called_once_with('foo', bar='baz')
>>> mock('other', bar='values')
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> mock.assert_called_once_with('other', bar='values')
Traceback (most recent call last):
  ...
AssertionError: Expected 'mock' to be called once. Called 2 times.
Calls: [call('foo', bar='baz'), call('other', bar='values')].
```

assert_any_call(**args*, ***kwargs*)

Asserter que le simulacre a été appelé avec les arguments spécifiés.

Asserter que le simulacre a *bien* été appelé avec les arguments au cours de la vie du simulacre. Contrairement à `assert_called_with()` et `assert_called_once_with()` qui passent seulement si l'appel demandé correspond bien au dernier appel, et dans le cas de `assert_called_once_with()` l'appel au simulacre doit être unique.

```
>>> mock = Mock(return_value=None)
>>> mock(1, 2, arg='thing')
>>> mock('some', 'thing', 'else')
>>> mock.assert_any_call(1, 2, arg='thing')
```

assert_has_calls(*calls*, *any_order=False*)

Asserter que le simulacre a été appelé avec les appels spécifiés. L'attribut `mock_calls` est comparé à la liste des appels.

If *any_order* is false then the calls must be sequential. There can be extra calls before or after the specified calls.

Si *any_order* est vrai alors les appels peuvent être dans n'importe quel ordre, mais ils doivent tous apparaître dans `mock_calls`.

```
>>> mock = Mock(return_value=None)
>>> mock(1)
>>> mock(2)
>>> mock(3)
>>> mock(4)
>>> calls = [call(2), call(3)]
>>> mock.assert_has_calls(calls)
>>> calls = [call(4), call(2), call(3)]
>>> mock.assert_has_calls(calls, any_order=True)
```

assert_not_called()

Asserter que le simulacre n'a jamais été appelé.

```
>>> m = Mock()
>>> m.hello.assert_not_called()
>>> obj = m.hello()
>>> m.hello.assert_not_called()
Traceback (most recent call last):
  ...
AssertionError: Expected 'hello' to not have been called. Called 1 times.
Calls: [call()].
```

Ajouté dans la version 3.5.

reset_mock(**, return_value=False, side_effect=False*)

La méthode *reset_mock* réinitialise tous les attributs d'appel sur un simulacre :

```
>>> mock = Mock(return_value=None)
>>> mock('hello')
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> mock.called
True
>>> mock.reset_mock()
>>> mock.called
False
```

This can be useful where you want to make a series of assertions that reuse the same object.

return_value parameter when set to `True` resets `return_value` :

```
>>> mock = Mock(return_value=5)
>>> mock('hello')
5
>>> mock.reset_mock(return_value=True)
>>> mock('hello')
<Mock name='mock()' id='...'>
```

side_effect parameter when set to `True` resets `side_effect` :

```
>>> mock = Mock(side_effect=ValueError)
>>> mock('hello')
Traceback (most recent call last):
  ...
ValueError
>>> mock.reset_mock(side_effect=True)
>>> mock('hello')
<Mock name='mock()' id='...'>
```

Note that `reset_mock()` *doesn't* clear the `return_value`, `side_effect` or any child attributes you have set using normal assignment by default.

Child mocks are reset as well.

Modifié dans la version 3.6 : Added two keyword-only arguments to the reset_mock function.

mock_add_spec(*spec*, *spec_set=False*)

Ajoute une spécification à un simulacre. *spec* peut être un objet ou une liste de chaînes de caractères. Seuls les attributs de la spécification *spec* peuvent être récupérés en tant qu'attributs du simulacre.

Si *spec_set* est vrai, seuls les attributs de la spécification peuvent être définis.

attach_mock(*mock*, *attribute*)

Attache un simulacre comme attribut de l'instance courante, en remplaçant son nom et son parent. Les appels au simulacre attaché sont enregistrés dans les attributs `method_calls` et `mock_calls` de l'instance courante.

configure_mock(***kwargs*)

Définir les attributs sur le simulacre à l'aide d'arguments nommés.

Les attributs, les valeurs de retour et les effets de bords peuvent être définis sur des simulacres enfants en utilisant la notation par points standard et en dépaquetant un dictionnaire dans l'appel de méthode :

```
>>> mock = Mock()
>>> attrs = {'method.return_value': 3, 'other.side_effect': KeyError}
>>> mock.configure_mock(**attrs)
>>> mock.method()
3
>>> mock.other()
Traceback (most recent call last):
  ...
KeyError
```

La même chose peut être réalisée en utilisant le constructeur des simulacres :

```
>>> attrs = {'method.return_value': 3, 'other.side_effect': KeyError}
>>> mock = Mock(some_attribute='eggs', **attrs)
>>> mock.some_attribute
'eggs'
>>> mock.method()
3
>>> mock.other()
Traceback (most recent call last):
  ...
KeyError
```

configure_mock() existe pour faciliter la configuration après la création du simulacre.

__dir__()

Les objets *Mock* limitent les résultats de `dir(un_mock)` à des résultats utiles. Pour les simulacres avec une spécification *spec*, cela inclut tous les attributs autorisés du simulacre.

Voir *FILTER_DIR* pour savoir ce que fait ce filtrage, et comment le désactiver.

_get_child_mock(***kw*)

Crée les simulacres enfants pour les attributs et la valeur de retour. Par défaut, les objets simulacre enfants sont du même type que le parent. Les sous-classes de *Mock* peuvent surcharger cette méthode pour personnaliser la façon dont les simulacres enfants sont créés.

Pour les simulacres non appelables, la variante appelable est utilisée (plutôt qu'une sous-classe personnalisée).

called

Un booléen représentant si le simulacre a bien été appelé ou non :

```
>>> mock = Mock(return_value=None)
>>> mock.called
False
>>> mock()
>>> mock.called
True
```

call_count

Un entier indiquant combien de fois le simulacre a été appelé :

```
>>> mock = Mock(return_value=None)
>>> mock.call_count
0
>>> mock()
>>> mock()
>>> mock.call_count
2
```

return_value

Définir cette option pour configurer la valeur renvoyé par appel du simulacre :

```
>>> mock = Mock()
>>> mock.return_value = 'fish'
>>> mock()
'fish'
```

La valeur de revoie par défaut est un simulacre configurable normalement :

```
>>> mock = Mock()
>>> mock.return_value.attribute = sentinel.Attribute
>>> mock.return_value()
<Mock name='mock()()' id='...'>
>>> mock.return_value.assert_called_with()
```

L'attribut `return_value` peut également être défini dans le constructeur :

```
>>> mock = Mock(return_value=3)
>>> mock.return_value
3
>>> mock()
3
```

side_effect

C'est soit une fonction à appeler lors de l'appel du simulacre, soit une exception (classe ou instance) à lever.

Si vous passez une fonction, elle est appelée avec les mêmes arguments que la fonction simulée et à moins que la fonction ne renvoie le singleton `DEFAULT` l'appel le la fonction simulée renvoie ce que la fonction renvoie. Si la fonction renvoie `DEFAULT` alors le simulacre renvoie sa valeur normale (celle de `return_value`).

Si vous passez un itérable, il est utilisé pour récupérer un itérateur qui doit renvoyer une valeur à chaque appel. Cette valeur peut être soit une instance d'exception à lever, soit une valeur à renvoyer à l'appel au simulacre (le traitement `DEFAULT` est identique au renvoie de la fonction simulée).

Un exemple d'un simulacre qui lève une exception (pour tester la gestion des exceptions d'une API) :

```
>>> mock = Mock()
>>> mock.side_effect = Exception('Boom!')
>>> mock()
Traceback (most recent call last):
  ...
Exception: Boom!
```

Utiliser `side_effect` pour renvoyer une séquence de valeurs :

```
>>> mock = Mock()
>>> mock.side_effect = [3, 2, 1]
>>> mock(), mock(), mock()
(3, 2, 1)
```

Utilisation d'un objet appelable :

```
>>> mock = Mock(return_value=3)
>>> def side_effect(*args, **kwargs):
...     return DEFAULT
...
>>> mock.side_effect = side_effect
>>> mock()
3
```

Un attribut `side_effect` peut être défini dans le constructeur. Voici un exemple qui ajoute un à la valeur du simulacre appelé et qui le renvoie :

```
>>> side_effect = lambda value: value + 1
>>> mock = Mock(side_effect=side_effect)
>>> mock(3)
```

(suite sur la page suivante)

(suite de la page précédente)

```
4
>>> mock(-8)
-7
```

Positionner *side_effect* sur `None` l'efface :

```
>>> m = Mock(side_effect=KeyError, return_value=3)
>>> m()
Traceback (most recent call last):
 ...
KeyError
>>> m.side_effect = None
>>> m()
3
```

call_args

This is either `None` (if the mock hasn't been called), or the arguments that the mock was last called with. This will be in the form of a tuple : the first member, which can also be accessed through the `args` property, is any ordered arguments the mock was called with (or an empty tuple) and the second member, which can also be accessed through the `kwargs` property, is any keyword arguments (or an empty dictionary).

```
>>> mock = Mock(return_value=None)
>>> print(mock.call_args)
None
>>> mock()
>>> mock.call_args
call()
>>> mock.call_args == ()
True
>>> mock(3, 4)
>>> mock.call_args
call(3, 4)
>>> mock.call_args == ((3, 4),)
True
>>> mock.call_args.args
(3, 4)
>>> mock.call_args.kwargs
{}
>>> mock(3, 4, 5, key='fish', next='w00t!')
>>> mock.call_args
call(3, 4, 5, key='fish', next='w00t!')
>>> mock.call_args.args
(3, 4, 5)
>>> mock.call_args.kwargs
{'key': 'fish', 'next': 'w00t!'}
```

L'attribut *call_args*, ainsi que les éléments des listes *call_args_list*, *method_calls* et *mock_calls* sont des objets *call*. Ce sont des *n*-uplets, que l'on peut dépaqueter afin de faire des affirmations plus complexes sur chacun des arguments. Voir *appels comme *n*-uplets*.

Modifié dans la version 3.8 : Added `args` and `kwargs` properties.

call_args_list

This is a list of all the calls made to the mock object in sequence (so the length of the list is the number of times it has been called). Before any calls have been made it is an empty list. The *call* object can be used for conveniently constructing lists of calls to compare with *call_args_list*.

```
>>> mock = Mock(return_value=None)
>>> mock()
>>> mock(3, 4)
>>> mock(key='fish', next='w00t!')
>>> mock.call_args_list
[call(), call(3, 4), call(key='fish', next='w00t!')]
>>> expected = [(), ((3, 4),), ({'key': 'fish', 'next': 'w00t!'},)]
>>> mock.call_args_list == expected
True
```

Members of `call_args_list` are `call` objects. These can be unpacked as tuples to get at the individual arguments. See *calls as tuples*.

method_calls

As well as tracking calls to themselves, mocks also track calls to methods and attributes, and *their* methods and attributes :

```
>>> mock = Mock()
>>> mock.method()
<Mock name='mock.method()' id='...'>
>>> mock.property.method.attribute()
<Mock name='mock.property.method.attribute()' id='...'>
>>> mock.method_calls
[call.method(), call.property.method.attribute()]
```

Members of `method_calls` are `call` objects. These can be unpacked as tuples to get at the individual arguments. See *calls as tuples*.

mock_calls

`mock_calls` records *all* calls to the mock object, its methods, magic methods *and* return value mocks.

```
>>> mock = MagicMock()
>>> result = mock(1, 2, 3)
>>> mock.first(a=3)
<MagicMock name='mock.first()' id='...'>
>>> mock.second()
<MagicMock name='mock.second()' id='...'>
>>> int(mock)
1
>>> result(1)
<MagicMock name='mock()()' id='...'>
>>> expected = [call(1, 2, 3), call.first(a=3), call.second(),
... call.__int__(), call()(1)]
>>> mock.mock_calls == expected
True
```

Members of `mock_calls` are `call` objects. These can be unpacked as tuples to get at the individual arguments. See *calls as tuples*.

Note

The way `mock_calls` are recorded means that where nested calls are made, the parameters of ancestor calls are not recorded and so will always compare equal :

```
>>> mock = MagicMock()
>>> mock.top(a=3).bottom()
<MagicMock name='mock.top().bottom()' id='...'>
>>> mock.mock_calls
[call.top(a=3), call.top().bottom()]
```

```
>>> mock.mock_calls[-1] == call.top(a=-1).bottom()
True
```

__class__

Normally the `__class__` attribute of an object will return its type. For a mock object with a `spec`, `__class__` returns the spec class instead. This allows mock objects to pass `isinstance()` tests for the object they are replacing / masquerading as :

```
>>> mock = Mock(spec=3)
>>> isinstance(mock, int)
True
```

`__class__` is assignable to, this allows a mock to pass an `isinstance()` check without forcing you to use a spec :

```
>>> mock = Mock()
>>> mock.__class__ = dict
>>> isinstance(mock, dict)
True
```

class `unittest.mock.`**NonCallableMock**(*spec=None*, *wraps=None*, *name=None*, *spec_set=None*, ***kwargs*)

A non-callable version of `Mock`. The constructor parameters have the same meaning of `Mock`, with the exception of *return_value* and *side_effect* which have no meaning on a non-callable mock.

Mock objects that use a class or an instance as a `spec` or `spec_set` are able to pass `isinstance()` tests :

```
>>> mock = Mock(spec=SomeClass)
>>> isinstance(mock, SomeClass)
True
>>> mock = Mock(spec_set=SomeClass())
>>> isinstance(mock, SomeClass)
True
```

The `Mock` classes have support for mocking magic methods. See *magic methods* for the full details.

The mock classes and the `patch()` decorators all take arbitrary keyword arguments for configuration. For the `patch()` decorators the keywords are passed to the constructor of the mock being created. The keyword arguments are for configuring attributes of the mock :

```
>>> m = MagicMock(attribute=3, other='fish')
>>> m.attribute
3
>>> m.other
'fish'
```

The return value and side effect of child mocks can be set in the same way, using dotted notation. As you can't use dotted names directly in a call you have to create a dictionary and unpack it using `**` :

```
>>> attrs = {'method.return_value': 3, 'other.side_effect': KeyError}
>>> mock = Mock(some_attribute='eggs', **attrs)
>>> mock.some_attribute
'eggs'
>>> mock.method()
3
>>> mock.other()
Traceback (most recent call last):
  ...
KeyError
```

A callable mock which was created with a *spec* (or a *spec_set*) will introspect the specification object's signature when matching calls to the mock. Therefore, it can match the actual call's arguments regardless of whether they were passed positionally or by name :

```
>>> def f(a, b, c): pass
...
>>> mock = Mock(spec=f)
>>> mock(1, 2, c=3)
<Mock name='mock()' id='140161580456576'>
>>> mock.assert_called_with(1, 2, 3)
>>> mock.assert_called_with(a=1, b=2, c=3)
```

This applies to `assert_called_with()`, `assert_called_once_with()`, `assert_has_calls()` and `assert_any_call()`. When *Autospeccing*, it will also apply to method calls on the mock object.

Modifié dans la version 3.4 : Added signature introspection on specced and autospecced mock objects.

class unittest.mock.**PropertyMock**(**args*, ***kwargs*)

A mock intended to be used as a `property`, or other *descriptor*, on a class. `PropertyMock` provides `__get__()` and `__set__()` methods so you can specify a return value when it is fetched.

Fetching a `PropertyMock` instance from an object calls the mock, with no args. Setting it calls the mock with the value being set.

```
>>> class Foo:
...     @property
...     def foo(self):
...         return 'something'
...     @foo.setter
...     def foo(self, value):
...         pass
...
>>> with patch('__main__.Foo.foo', new_callable=PropertyMock) as mock_foo:
...     mock_foo.return_value = 'mockity-mock'
...     this_foo = Foo()
...     print(this_foo.foo)
...     this_foo.foo = 6
...
mockity-mock
>>> mock_foo.mock_calls
[call(), call(6)]
```

Because of the way mock attributes are stored you can't directly attach a `PropertyMock` to a mock object. Instead you can attach it to the mock type object :

```
>>> m = MagicMock()
>>> p = PropertyMock(return_value=3)
>>> type(m).foo = p
>>> m.foo
3
>>> p.assert_called_once_with()
```

Prudence

If an `AttributeError` is raised by `PropertyMock`, it will be interpreted as a missing descriptor and `__getattr__()` will be called on the parent mock :

```
>>> m = MagicMock()
>>> no_attribute = PropertyMock(side_effect=AttributeError)
>>> type(m).my_property = no_attribute
```

```
>>> m.my_property
<MagicMock name='mock.my_property' id='140165240345424'>
```

See `__getattr__()` for details.

class unittest.mock.**AsyncMock**(*spec=None*, *side_effect=None*, *return_value=DEFAULT*, *wraps=None*, *name=None*, *spec_set=None*, *unsafe=False*, ***kwargs*)

An asynchronous version of `MagicMock`. The `AsyncMock` object will behave so the object is recognized as an async function, and the result of a call is an awaitable.

```
>>> mock = AsyncMock()
>>> inspect.iscoroutinefunction(mock)
True
>>> inspect.isawaitable(mock())
True
```

The result of `mock()` is an async function which will have the outcome of `side_effect` or `return_value` after it has been awaited :

— if `side_effect` is a function, the async function will return the result of that function,
— if `side_effect` is an exception, the async function will raise the exception,
— if `side_effect` is an iterable, the async function will return the next value of the iterable, however, if the sequence of result is exhausted, `StopAsyncIteration` is raised immediately,
— if `side_effect` is not defined, the async function will return the value defined by `return_value`, hence, by default, the async function returns a new `AsyncMock` object.

Setting the *spec* of a `Mock` or `MagicMock` to an async function will result in a coroutine object being returned after calling.

```
>>> async def async_func(): pass
...
>>> mock = MagicMock(async_func)
>>> mock
<MagicMock spec='function' id='...'>
>>> mock()
<coroutine object AsyncMockMixin._mock_call at ...>
```

Setting the *spec* of a `Mock`, `MagicMock`, or `AsyncMock` to a class with asynchronous and synchronous functions will automatically detect the synchronous functions and set them as `MagicMock` (if the parent mock is `AsyncMock` or `MagicMock`) or `Mock` (if the parent mock is `Mock`). All asynchronous functions will be `AsyncMock`.

```
>>> class ExampleClass:
...     def sync_foo():
...         pass
...     async def async_foo():
...         pass
...
>>> a_mock = AsyncMock(ExampleClass)
>>> a_mock.sync_foo
<MagicMock name='mock.sync_foo' id='...'>
>>> a_mock.async_foo
<AsyncMock name='mock.async_foo' id='...'>
>>> mock = Mock(ExampleClass)
>>> mock.sync_foo
<Mock name='mock.sync_foo' id='...'>
>>> mock.async_foo
<AsyncMock name='mock.async_foo' id='...'>
```

Ajouté dans la version 3.8.

assert_awaited()

Assert that the mock was awaited at least once. Note that this is separate from the object having been called, the `await` keyword must be used :

```
>>> mock = AsyncMock()
>>> async def main(coroutine_mock):
...     await coroutine_mock
...
>>> coroutine_mock = mock()
>>> mock.called
True
>>> mock.assert_awaited()
Traceback (most recent call last):
...
AssertionError: Expected mock to have been awaited.
>>> asyncio.run(main(coroutine_mock))
>>> mock.assert_awaited()
```

assert_awaited_once()

Assert that the mock was awaited exactly once.

```
>>> mock = AsyncMock()
>>> async def main():
...     await mock()
...
>>> asyncio.run(main())
>>> mock.assert_awaited_once()
>>> asyncio.run(main())
>>> mock.assert_awaited_once()
Traceback (most recent call last):
...
AssertionError: Expected mock to have been awaited once. Awaited 2 times.
```

assert_awaited_with(**args*, ***kwargs*)

Assert that the last await was with the specified arguments.

```
>>> mock = AsyncMock()
>>> async def main(*args, **kwargs):
...     await mock(*args, **kwargs)
...
>>> asyncio.run(main('foo', bar='bar'))
>>> mock.assert_awaited_with('foo', bar='bar')
>>> mock.assert_awaited_with('other')
Traceback (most recent call last):
...
AssertionError: expected await not found.
Expected: mock('other')
Actual: mock('foo', bar='bar')
```

assert_awaited_once_with(**args*, ***kwargs*)

Assert that the mock was awaited exactly once and with the specified arguments.

```
>>> mock = AsyncMock()
>>> async def main(*args, **kwargs):
...     await mock(*args, **kwargs)
...
>>> asyncio.run(main('foo', bar='bar'))
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> mock.assert_awaited_once_with('foo', bar='bar')
>>> asyncio.run(main('foo', bar='bar'))
>>> mock.assert_awaited_once_with('foo', bar='bar')
Traceback (most recent call last):
...
AssertionError: Expected mock to have been awaited once. Awaited 2 times.
```

assert_any_await(**args*, ***kwargs*)

Assert the mock has ever been awaited with the specified arguments.

```
>>> mock = AsyncMock()
>>> async def main(*args, **kwargs):
...     await mock(*args, **kwargs)
...
>>> asyncio.run(main('foo', bar='bar'))
>>> asyncio.run(main('hello'))
>>> mock.assert_any_await('foo', bar='bar')
>>> mock.assert_any_await('other')
Traceback (most recent call last):
...
AssertionError: mock('other') await not found
```

assert_has_awaits(*calls*, *any_order=False*)

Assert the mock has been awaited with the specified calls. The `await_args_list` list is checked for the awaits.

If *any_order* is false then the awaits must be sequential. There can be extra calls before or after the specified awaits.

If *any_order* is true then the awaits can be in any order, but they must all appear in `await_args_list`.

```
>>> mock = AsyncMock()
>>> async def main(*args, **kwargs):
...     await mock(*args, **kwargs)
...
>>> calls = [call("foo"), call("bar")]
>>> mock.assert_has_awaits(calls)
Traceback (most recent call last):
...
AssertionError: Awaits not found.
Expected: [call('foo'), call('bar')]
Actual: []
>>> asyncio.run(main('foo'))
>>> asyncio.run(main('bar'))
>>> mock.assert_has_awaits(calls)
```

assert_not_awaited()

Assert that the mock was never awaited.

```
>>> mock = AsyncMock()
>>> mock.assert_not_awaited()
```

reset_mock(**args*, ***kwargs*)

See `Mock.reset_mock()`. Also sets `await_count` to 0, `await_args` to None, and clears the `await_args_list`.

await_count

An integer keeping track of how many times the mock object has been awaited.

```
>>> mock = AsyncMock()
>>> async def main():
...     await mock()
...
>>> asyncio.run(main())
>>> mock.await_count
1
>>> asyncio.run(main())
>>> mock.await_count
2
```

await_args

This is either `None` (if the mock hasn't been awaited), or the arguments that the mock was last awaited with. Functions the same as `Mock.call_args`.

```
>>> mock = AsyncMock()
>>> async def main(*args):
...     await mock(*args)
...
>>> mock.await_args
>>> asyncio.run(main('foo'))
>>> mock.await_args
call('foo')
>>> asyncio.run(main('bar'))
>>> mock.await_args
call('bar')
```

await_args_list

This is a list of all the awaits made to the mock object in sequence (so the length of the list is the number of times it has been awaited). Before any awaits have been made it is an empty list.

```
>>> mock = AsyncMock()
>>> async def main(*args):
...     await mock(*args)
...
>>> mock.await_args_list
[]
>>> asyncio.run(main('foo'))
>>> mock.await_args_list
[call('foo')]
>>> asyncio.run(main('bar'))
>>> mock.await_args_list
[call('foo'), call('bar')]
```

class unittest.mock.**ThreadingMock**(*spec=None*, *side_effect=None*, *return_value=DEFAULT*, *wraps=None*, *name=None*, *spec_set=None*, *unsafe=False*, *, *timeout=UNSET*, ***kwargs*)

A version of `MagicMock` for multithreading tests. The `ThreadingMock` object provides extra methods to wait for a call to be invoked, rather than assert on it immediately.

The default timeout is specified by the `timeout` argument, or if unset by the `ThreadingMock.DEFAULT_TIMEOUT` attribute, which defaults to blocking (`None`).

You can configure the global default timeout by setting `ThreadingMock.DEFAULT_TIMEOUT`.

wait_until_called(*, *timeout=UNSET*)

Waits until the mock is called.

If a timeout was passed at the creation of the mock or if a timeout argument is passed to this function,

the function raises an `AssertionError` if the call is not performed in time.

```
>>> mock = ThreadingMock()
>>> thread = threading.Thread(target=mock)
>>> thread.start()
>>> mock.wait_until_called(timeout=1)
>>> thread.join()
```

wait_until_any_call_with(**args*, ***kwargs*)

Waits until the mock is called with the specified arguments.

If a timeout was passed at the creation of the mock the function raises an `AssertionError` if the call is not performed in time.

```
>>> mock = ThreadingMock()
>>> thread = threading.Thread(target=mock, args=("arg1", "arg2",), kwargs={
↪"arg": "thing"})
>>> thread.start()
>>> mock.wait_until_any_call_with("arg1", "arg2", arg="thing")
>>> thread.join()
```

DEFAULT_TIMEOUT

Global default timeout in seconds to create instances of `ThreadingMock`.

Ajouté dans la version 3.13.

Calling

Mock objects are callable. The call will return the value set as the `return_value` attribute. The default return value is a new Mock object ; it is created the first time the return value is accessed (either explicitly or by calling the Mock) - but it is stored and the same one returned each time.

Calls made to the object will be recorded in the attributes like `call_args` and `call_args_list`.

If `side_effect` is set then it will be called after the call has been recorded, so if `side_effect` raises an exception the call is still recorded.

The simplest way to make a mock raise an exception when called is to make `side_effect` an exception class or instance :

```
>>> m = MagicMock(side_effect=IndexError)
>>> m(1, 2, 3)
Traceback (most recent call last):
  ...
IndexError
>>> m.mock_calls
[call(1, 2, 3)]
>>> m.side_effect = KeyError('Bang!')
>>> m('two', 'three', 'four')
Traceback (most recent call last):
  ...
KeyError: 'Bang!'
>>> m.mock_calls
[call(1, 2, 3), call('two', 'three', 'four')]
```

If `side_effect` is a function then whatever that function returns is what calls to the mock return. The `side_effect` function is called with the same arguments as the mock. This allows you to vary the return value of the call dynamically, based on the input :

```
>>> def side_effect(value):
...     return value + 1
```

(suite sur la page suivante)

(suite de la page précédente)

```
...
>>> m = MagicMock(side_effect=side_effect)
>>> m(1)
2
>>> m(2)
3
>>> m.mock_calls
[call(1), call(2)]
```

If you want the mock to still return the default return value (a new mock), or any set return value, then there are two ways of doing this. Either return *return_value* from inside *side_effect*, or return *DEFAULT* :

```
>>> m = MagicMock()
>>> def side_effect(*args, **kwargs):
...     return m.return_value
...
>>> m.side_effect = side_effect
>>> m.return_value = 3
>>> m()
3
>>> def side_effect(*args, **kwargs):
...     return DEFAULT
...
>>> m.side_effect = side_effect
>>> m()
3
```

To remove a *side_effect*, and return to the default behaviour, set the `side_effect` to `None` :

```
>>> m = MagicMock(return_value=6)
>>> def side_effect(*args, **kwargs):
...     return 3
...
>>> m.side_effect = side_effect
>>> m()
3
>>> m.side_effect = None
>>> m()
6
```

The *side_effect* can also be any iterable object. Repeated calls to the mock will return values from the iterable (until the iterable is exhausted and a *StopIteration* is raised) :

```
>>> m = MagicMock(side_effect=[1, 2, 3])
>>> m()
1
>>> m()
2
>>> m()
3
>>> m()
Traceback (most recent call last):
  ...
StopIteration
```

If any members of the iterable are exceptions they will be raised instead of returned :

```
>>> iterable = (33, ValueError, 66)
>>> m = MagicMock(side_effect=iterable)
>>> m()
33
>>> m()
Traceback (most recent call last):
 ...
ValueError
>>> m()
66
```

Deleting Attributes

Mock objects create attributes on demand. This allows them to pretend to be objects of any type.

You may want a mock object to return `False` to a *hasattr()* call, or raise an *AttributeError* when an attribute is fetched. You can do this by providing an object as a `spec` for a mock, but that isn't always convenient.

You "block" attributes by deleting them. Once deleted, accessing an attribute will raise an *AttributeError*.

```
>>> mock = MagicMock()
>>> hasattr(mock, 'm')
True
>>> del mock.m
>>> hasattr(mock, 'm')
False
>>> del mock.f
>>> mock.f
Traceback (most recent call last):
    ...
AttributeError: f
```

Mock names and the name attribute

Since "name" is an argument to the *Mock* constructor, if you want your mock object to have a "name" attribute you can't just pass it in at creation time. There are two alternatives. One option is to use *configure_mock()* :

```
>>> mock = MagicMock()
>>> mock.configure_mock(name='my_name')
>>> mock.name
'my_name'
```

A simpler option is to simply set the "name" attribute after mock creation :

```
>>> mock = MagicMock()
>>> mock.name = "foo"
```

Attaching Mocks as Attributes

When you attach a mock as an attribute of another mock (or as the return value) it becomes a "child" of that mock. Calls to the child are recorded in the *method_calls* and *mock_calls* attributes of the parent. This is useful for configuring child mocks and then attaching them to the parent, or for attaching mocks to a parent that records all calls to the children and allows you to make assertions about the order of calls between mocks :

```
>>> parent = MagicMock()
>>> child1 = MagicMock(return_value=None)
>>> child2 = MagicMock(return_value=None)
>>> parent.child1 = child1
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> parent.child2 = child2
>>> child1(1)
>>> child2(2)
>>> parent.mock_calls
[call.child1(1), call.child2(2)]
```

The exception to this is if the mock has a name. This allows you to prevent the "parenting" if for some reason you don't want it to happen.

```
>>> mock = MagicMock()
>>> not_a_child = MagicMock(name='not-a-child')
>>> mock.attribute = not_a_child
>>> mock.attribute()
<MagicMock name='not-a-child()' id='...'>
>>> mock.mock_calls
[]
```

Mocks created for you by `patch()` are automatically given names. To attach mocks that have names to a parent you use the `attach_mock()` method :

```
>>> thing1 = object()
>>> thing2 = object()
>>> parent = MagicMock()
>>> with patch('__main__.thing1', return_value=None) as child1:
...     with patch('__main__.thing2', return_value=None) as child2:
...         parent.attach_mock(child1, 'child1')
...         parent.attach_mock(child2, 'child2')
...         child1('one')
...         child2('two')
...
>>> parent.mock_calls
[call.child1('one'), call.child2('two')]
```

27.6.3 The patchers

The patch decorators are used for patching objects only within the scope of the function they decorate. They automatically handle the unpatching for you, even if exceptions are raised. All of these functions can also be used in with statements or as class decorators.

patch

> **Note**
>
> The key is to do the patching in the right namespace. See the section *where to patch*.

`unittest.mock.patch`(*target*, *new=DEFAULT*, *spec=None*, *create=False*, *spec_set=None*, *autospec=None*, *new_callable=None*, ***kwargs*)

`patch()` acts as a function decorator, class decorator or a context manager. Inside the body of the function or with statement, the *target* is patched with a *new* object. When the function/with statement exits the patch is undone.

If *new* is omitted, then the target is replaced with an `AsyncMock` if the patched object is an async function or a `MagicMock` otherwise. If `patch()` is used as a decorator and *new* is omitted, the created mock is passed in as an extra argument to the decorated function. If `patch()` is used as a context manager the created mock is returned by the context manager.

target should be a string in the form `'package.module.ClassName'`. The *target* is imported and the specified object replaced with the *new* object, so the *target* must be importable from the environment you are calling `patch()` from. The target is imported when the decorated function is executed, not at decoration time.

The *spec* and *spec_set* keyword arguments are passed to the `MagicMock` if patch is creating one for you.

In addition you can pass `spec=True` or `spec_set=True`, which causes patch to pass in the object being mocked as the spec/spec_set object.

new_callable allows you to specify a different class, or callable object, that will be called to create the *new* object. By default `AsyncMock` is used for async functions and `MagicMock` for the rest.

A more powerful form of *spec* is *autospec*. If you set `autospec=True` then the mock will be created with a spec from the object being replaced. All attributes of the mock will also have the spec of the corresponding attribute of the object being replaced. Methods and functions being mocked will have their arguments checked and will raise a `TypeError` if they are called with the wrong signature. For mocks replacing a class, their return value (the 'instance') will have the same spec as the class. See the `create_autospec()` function and *Autospeccing*.

Instead of `autospec=True` you can pass `autospec=some_object` to use an arbitrary object as the spec instead of the one being replaced.

By default `patch()` will fail to replace attributes that don't exist. If you pass in `create=True`, and the attribute doesn't exist, patch will create the attribute for you when the patched function is called, and delete it again after the patched function has exited. This is useful for writing tests against attributes that your production code creates at runtime. It is off by default because it can be dangerous. With it switched on you can write passing tests against APIs that don't actually exist !

> **Note**
>
> Modifié dans la version 3.5 : If you are patching builtins in a module then you don't need to pass `create=True`, it will be added by default.

Patch can be used as a `TestCase` class decorator. It works by decorating each test method in the class. This reduces the boilerplate code when your test methods share a common patchings set. `patch()` finds tests by looking for method names that start with `patch.TEST_PREFIX`. By default this is `'test'`, which matches the way `unittest` finds tests. You can specify an alternative prefix by setting `patch.TEST_PREFIX`.

Patch can be used as a context manager, with the with statement. Here the patching applies to the indented block after the with statement. If you use "as" then the patched object will be bound to the name after the "as" ; very useful if `patch()` is creating a mock object for you.

`patch()` takes arbitrary keyword arguments. These will be passed to `AsyncMock` if the patched object is asynchronous, to `MagicMock` otherwise or to *new_callable* if specified.

`patch.dict(...)`, `patch.multiple(...)` and `patch.object(...)` are available for alternate use-cases.

`patch()` as function decorator, creating the mock for you and passing it into the decorated function :

```
>>> @patch('__main__.SomeClass')
... def function(normal_argument, mock_class):
...     print(mock_class is SomeClass)
...
>>> function(None)
True
```

Patching a class replaces the class with a `MagicMock` *instance*. If the class is instantiated in the code under test then it will be the `return_value` of the mock that will be used.

If the class is instantiated multiple times you could use `side_effect` to return a new mock each time. Alternatively you can set the *return_value* to be anything you want.

To configure return values on methods of *instances* on the patched class you must do this on the `return_value`. For example :

```
>>> class Class:
...     def method(self):
...         pass
...
>>> with patch('__main__.Class') as MockClass:
...     instance = MockClass.return_value
...     instance.method.return_value = 'foo'
...     assert Class() is instance
...     assert Class().method() == 'foo'
...
```

If you use *spec* or *spec_set* and `patch()` is replacing a *class*, then the return value of the created mock will have the same spec.

```
>>> Original = Class
>>> patcher = patch('__main__.Class', spec=True)
>>> MockClass = patcher.start()
>>> instance = MockClass()
>>> assert isinstance(instance, Original)
>>> patcher.stop()
```

The *new_callable* argument is useful where you want to use an alternative class to the default `MagicMock` for the created mock. For example, if you wanted a `NonCallableMock` to be used :

```
>>> thing = object()
>>> with patch('__main__.thing', new_callable=NonCallableMock) as mock_thing:
...     assert thing is mock_thing
...     thing()
...
Traceback (most recent call last):
  ...
TypeError: 'NonCallableMock' object is not callable
```

Another use case might be to replace an object with an `io.StringIO` instance :

```
>>> from io import StringIO
>>> def foo():
...     print('Something')
...
>>> @patch('sys.stdout', new_callable=StringIO)
... def test(mock_stdout):
...     foo()
...     assert mock_stdout.getvalue() == 'Something\n'
...
>>> test()
```

When `patch()` is creating a mock for you, it is common that the first thing you need to do is to configure the mock. Some of that configuration can be done in the call to patch. Any arbitrary keywords you pass into the call will be used to set attributes on the created mock :

```
>>> patcher = patch('__main__.thing', first='one', second='two')
>>> mock_thing = patcher.start()
>>> mock_thing.first
'one'
>>> mock_thing.second
'two'
```

As well as attributes on the created mock attributes, like the `return_value` and `side_effect`, of child mocks can also be configured. These aren't syntactically valid to pass in directly as keyword arguments, but a dictionary with these as keys can still be expanded into a `patch()` call using `**` :

```
>>> config = {'method.return_value': 3, 'other.side_effect': KeyError}
>>> patcher = patch('__main__.thing', **config)
>>> mock_thing = patcher.start()
>>> mock_thing.method()
3
>>> mock_thing.other()
Traceback (most recent call last):
  ...
KeyError
```

By default, attempting to patch a function in a module (or a method or an attribute in a class) that does not exist will fail with `AttributeError` :

```
>>> @patch('sys.non_existing_attribute', 42)
... def test():
...     assert sys.non_existing_attribute == 42
...
>>> test()
Traceback (most recent call last):
  ...
AttributeError: <module 'sys' (built-in)> does not have the attribute 'non_
↪existing_attribute'
```

but adding `create=True` in the call to `patch()` will make the previous example work as expected :

```
>>> @patch('sys.non_existing_attribute', 42, create=True)
... def test(mock_stdout):
...     assert sys.non_existing_attribute == 42
...
>>> test()
```

Modifié dans la version 3.8 : `patch()` now returns an `AsyncMock` if the target is an async function.

patch.object

`patch.object`(*target*, *attribute*, *new=DEFAULT*, *spec=None*, *create=False*, *spec_set=None*, *autospec=None*, *new_callable=None*, ***kwargs*)

patch the named member (*attribute*) on an object (*target*) with a mock object.

`patch.object()` can be used as a decorator, class decorator or a context manager. Arguments *new*, *spec*, *create*, *spec_set*, *autospec* and *new_callable* have the same meaning as for `patch()`. Like `patch()`, `patch.object()` takes arbitrary keyword arguments for configuring the mock object it creates.

When used as a class decorator `patch.object()` honours `patch.TEST_PREFIX` for choosing which methods to wrap.

You can either call `patch.object()` with three arguments or two arguments. The three argument form takes the object to be patched, the attribute name and the object to replace the attribute with.

When calling with the two argument form you omit the replacement object, and a mock is created for you and passed in as an extra argument to the decorated function :

```
>>> @patch.object(SomeClass, 'class_method')
... def test(mock_method):
...     SomeClass.class_method(3)
...     mock_method.assert_called_with(3)
```

(suite sur la page suivante)

(suite de la page précédente)

```
...
>>> test()
```

spec, *create* and the other arguments to `patch.object()` have the same meaning as they do for `patch()`.

patch.dict

`patch.dict`(*in_dict*, *values=()*, *clear=False*, ***kwargs*)

Patch a dictionary, or dictionary like object, and restore the dictionary to its original state after the test, where the restored dictionary is a copy of the dictionary as it was before the test.

in_dict can be a dictionary or a mapping like container. If it is a mapping then it must at least support getting, setting and deleting items plus iterating over keys.

in_dict can also be a string specifying the name of the dictionary, which will then be fetched by importing it.

values can be a dictionary of values to set in the dictionary. *values* can also be an iterable of `(key, value)` pairs.

If *clear* is true then the dictionary will be cleared before the new values are set.

`patch.dict()` can also be called with arbitrary keyword arguments to set values in the dictionary.

Modifié dans la version 3.8 : `patch.dict()` now returns the patched dictionary when used as a context manager.

`patch.dict()` can be used as a context manager, decorator or class decorator :

```
>>> foo = {}
>>> @patch.dict(foo, {'newkey': 'newvalue'})
... def test():
...     assert foo == {'newkey': 'newvalue'}
...
>>> test()
>>> assert foo == {}
```

When used as a class decorator `patch.dict()` honours `patch.TEST_PREFIX` (default to `'test'`) for choosing which methods to wrap :

```
>>> import os
>>> import unittest
>>> from unittest.mock import patch
>>> @patch.dict('os.environ', {'newkey': 'newvalue'})
... class TestSample(unittest.TestCase):
...     def test_sample(self):
...         self.assertEqual(os.environ['newkey'], 'newvalue')
```

If you want to use a different prefix for your test, you can inform the patchers of the different prefix by setting `patch.TEST_PREFIX`. For more details about how to change the value of see *TEST_PREFIX*.

`patch.dict()` can be used to add members to a dictionary, or simply let a test change a dictionary, and ensure the dictionary is restored when the test ends.

```
>>> foo = {}
>>> with patch.dict(foo, {'newkey': 'newvalue'}) as patched_foo:
...     assert foo == {'newkey': 'newvalue'}
...     assert patched_foo == {'newkey': 'newvalue'}
...     # You can add, update or delete keys of foo (or patched_foo, it's the same␣
↪dict)
...     patched_foo['spam'] = 'eggs'
...
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> assert foo == {}
>>> assert patched_foo == {}
```

```
>>> import os
>>> with patch.dict('os.environ', {'newkey': 'newvalue'}):
...     print(os.environ['newkey'])
...
newvalue
>>> assert 'newkey' not in os.environ
```

Keywords can be used in the `patch.dict()` call to set values in the dictionary :

```
>>> mymodule = MagicMock()
>>> mymodule.function.return_value = 'fish'
>>> with patch.dict('sys.modules', mymodule=mymodule):
...     import mymodule
...     mymodule.function('some', 'args')
...
'fish'
```

`patch.dict()` can be used with dictionary like objects that aren't actually dictionaries. At the very minimum they must support item getting, setting, deleting and either iteration or membership test. This corresponds to the magic methods `__getitem__()`, `__setitem__()`, `__delitem__()` and either `__iter__()` or `__contains__()`.

```
>>> class Container:
...     def __init__(self):
...         self.values = {}
...     def __getitem__(self, name):
...         return self.values[name]
...     def __setitem__(self, name, value):
...         self.values[name] = value
...     def __delitem__(self, name):
...         del self.values[name]
...     def __iter__(self):
...         return iter(self.values)
...
>>> thing = Container()
>>> thing['one'] = 1
>>> with patch.dict(thing, one=2, two=3):
...     assert thing['one'] == 2
...     assert thing['two'] == 3
...
>>> assert thing['one'] == 1
>>> assert list(thing) == ['one']
```

patch.multiple

`patch.multiple`(*target*, *spec=None*, *create=False*, *spec_set=None*, *autospec=None*, *new_callable=None*, ***kwargs*)

Perform multiple patches in a single call. It takes the object to be patched (either as an object or a string to fetch the object by importing) and keyword arguments for the patches :

```
with patch.multiple(settings, FIRST_PATCH='one', SECOND_PATCH='two'):
    ...
```

Use `DEFAULT` as the value if you want `patch.multiple()` to create mocks for you. In this case the created mocks are passed into a decorated function by keyword, and a dictionary is returned when `patch.`

multiple() is used as a context manager.

patch.multiple() can be used as a decorator, class decorator or a context manager. The arguments *spec*, *spec_set*, *create*, *autospec* and *new_callable* have the same meaning as for *patch()*. These arguments will be applied to *all* patches done by *patch.multiple()*.

When used as a class decorator *patch.multiple()* honours `patch.TEST_PREFIX` for choosing which methods to wrap.

If you want *patch.multiple()* to create mocks for you, then you can use *DEFAULT* as the value. If you use *patch.multiple()* as a decorator then the created mocks are passed into the decorated function by keyword.

```
>>> thing = object()
>>> other = object()

>>> @patch.multiple('__main__', thing=DEFAULT, other=DEFAULT)
... def test_function(thing, other):
...     assert isinstance(thing, MagicMock)
...     assert isinstance(other, MagicMock)
...
>>> test_function()
```

patch.multiple() can be nested with other `patch` decorators, but put arguments passed by keyword *after* any of the standard arguments created by *patch()* :

```
>>> @patch('sys.exit')
... @patch.multiple('__main__', thing=DEFAULT, other=DEFAULT)
... def test_function(mock_exit, other, thing):
...     assert 'other' in repr(other)
...     assert 'thing' in repr(thing)
...     assert 'exit' in repr(mock_exit)
...
>>> test_function()
```

If *patch.multiple()* is used as a context manager, the value returned by the context manager is a dictionary where created mocks are keyed by name :

```
>>> with patch.multiple('__main__', thing=DEFAULT, other=DEFAULT) as values:
...     assert 'other' in repr(values['other'])
...     assert 'thing' in repr(values['thing'])
...     assert values['thing'] is thing
...     assert values['other'] is other
...
```

patch methods : start and stop

All the patchers have `start()` and `stop()` methods. These make it simpler to do patching in `setUp` methods or where you want to do multiple patches without nesting decorators or with statements.

To use them call *patch()*, *patch.object()* or *patch.dict()* as normal and keep a reference to the returned `patcher` object. You can then call `start()` to put the patch in place and `stop()` to undo it.

If you are using *patch()* to create a mock for you then it will be returned by the call to `patcher.start`.

```
>>> patcher = patch('package.module.ClassName')
>>> from package import module
>>> original = module.ClassName
>>> new_mock = patcher.start()
>>> assert module.ClassName is not original
>>> assert module.ClassName is new_mock
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> patcher.stop()
>>> assert module.ClassName is original
>>> assert module.ClassName is not new_mock
```

A typical use case for this might be for doing multiple patches in the `setUp` method of a *TestCase* :

```
>>> class MyTest(unittest.TestCase):
...     def setUp(self):
...         self.patcher1 = patch('package.module.Class1')
...         self.patcher2 = patch('package.module.Class2')
...         self.MockClass1 = self.patcher1.start()
...         self.MockClass2 = self.patcher2.start()
...
...     def tearDown(self):
...         self.patcher1.stop()
...         self.patcher2.stop()
...
...     def test_something(self):
...         assert package.module.Class1 is self.MockClass1
...         assert package.module.Class2 is self.MockClass2
...
>>> MyTest('test_something').run()
```

Prudence

If you use this technique you must ensure that the patching is "undone" by calling `stop`. This can be fiddlier than you might think, because if an exception is raised in the `setUp` then `tearDown` is not called. *unittest.TestCase.addCleanup()* makes this easier :

```
>>> class MyTest(unittest.TestCase):
...     def setUp(self):
...         patcher = patch('package.module.Class')
...         self.MockClass = patcher.start()
...         self.addCleanup(patcher.stop)
...
...     def test_something(self):
...         assert package.module.Class is self.MockClass
...
```

As an added bonus you no longer need to keep a reference to the `patcher` object.

It is also possible to stop all patches which have been started by using *patch.stopall()*.

`patch.stopall()`

Stop all active patches. Only stops patches started with `start`.

patch builtins

You can patch any builtins within a module. The following example patches builtin *ord()* :

```
>>> @patch('__main__.ord')
... def test(mock_ord):
...     mock_ord.return_value = 101
...     print(ord('c'))
...
>>> test()
101
```

TEST_PREFIX

All of the patchers can be used as class decorators. When used in this way they wrap every test method on the class. The patchers recognise methods that start with `'test'` as being test methods. This is the same way that the *unittest.TestLoader* finds test methods by default.

It is possible that you want to use a different prefix for your tests. You can inform the patchers of the different prefix by setting `patch.TEST_PREFIX` :

```
>>> patch.TEST_PREFIX = 'foo'
>>> value = 3
>>>
>>> @patch('__main__.value', 'not three')
... class Thing:
...     def foo_one(self):
...         print(value)
...     def foo_two(self):
...         print(value)
...
>>>
>>> Thing().foo_one()
not three
>>> Thing().foo_two()
not three
>>> value
3
```

Nesting Patch Decorators

If you want to perform multiple patches then you can simply stack up the decorators.

You can stack up multiple patch decorators using this pattern :

```
>>> @patch.object(SomeClass, 'class_method')
... @patch.object(SomeClass, 'static_method')
... def test(mock1, mock2):
...     assert SomeClass.static_method is mock1
...     assert SomeClass.class_method is mock2
...     SomeClass.static_method('foo')
...     SomeClass.class_method('bar')
...     return mock1, mock2
...
>>> mock1, mock2 = test()
>>> mock1.assert_called_once_with('foo')
>>> mock2.assert_called_once_with('bar')
```

Note that the decorators are applied from the bottom upwards. This is the standard way that Python applies decorators. The order of the created mocks passed into your test function matches this order.

Where to patch

patch() works by (temporarily) changing the object that a *name* points to with another one. There can be many names pointing to any individual object, so for patching to work you must ensure that you patch the name used by the system under test.

The basic principle is that you patch where an object is *looked up*, which is not necessarily the same place as where it is defined. A couple of examples will help to clarify this.

Imagine we have a project that we want to test with the following structure :

```
a.py
    -> Defines SomeClass

b.py
    -> from a import SomeClass
    -> some_function instantiates SomeClass
```

Now we want to test `some_function` but we want to mock out `SomeClass` using *`patch()`*. The problem is that when we import module b, which we will have to do when it imports `SomeClass` from module a. If we use *`patch()`* to mock out `a.SomeClass` then it will have no effect on our test; module b already has a reference to the *real* `SomeClass` and it looks like our patching had no effect.

The key is to patch out `SomeClass` where it is used (or where it is looked up). In this case `some_function` will actually look up `SomeClass` in module b, where we have imported it. The patching should look like :

```
@patch('b.SomeClass')
```

However, consider the alternative scenario where instead of `from a import SomeClass` module b does `import a` and `some_function` uses `a.SomeClass`. Both of these import forms are common. In this case the class we want to patch is being looked up in the module and so we have to patch `a.SomeClass` instead :

```
@patch('a.SomeClass')
```

Patching Descriptors and Proxy Objects

Both *patch* and *patch.object* correctly patch and restore descriptors : class methods, static methods and properties. You should patch these on the *class* rather than an instance. They also work with *some* objects that proxy attribute access, like the django settings object.

27.6.4 MagicMock and magic method support

Mocking Magic Methods

`Mock` supports mocking the Python protocol methods, also known as *"magic methods"*. This allows mock objects to replace containers or other objects that implement Python protocols.

Because magic methods are looked up differently from normal methods [2], this support has been specially implemented. This means that only specific magic methods are supported. The supported list includes *almost* all of them. If there are any missing that you need please let us know.

You mock magic methods by setting the method you are interested in to a function or a mock instance. If you are using a function then it *must* take `self` as the first argument [3].

```
>>> def __str__(self):
...     return 'fooble'
...
>>> mock = Mock()
>>> mock.__str__ = __str__
>>> str(mock)
'fooble'
```

```
>>> mock = Mock()
>>> mock.__str__ = Mock()
>>> mock.__str__.return_value = 'fooble'
>>> str(mock)
'fooble'
```

2. Magic methods *should* be looked up on the class rather than the instance. Different versions of Python are inconsistent about applying this rule. The supported protocol methods should work with all supported versions of Python.
3. The function is basically hooked up to the class, but each `Mock` instance is kept isolated from the others.

```
>>> mock = Mock()
>>> mock.__iter__ = Mock(return_value=iter([]))
>>> list(mock)
[]
```

One use case for this is for mocking objects used as context managers in a `with` statement :

```
>>> mock = Mock()
>>> mock.__enter__ = Mock(return_value='foo')
>>> mock.__exit__ = Mock(return_value=False)
>>> with mock as m:
...     assert m == 'foo'
...
>>> mock.__enter__.assert_called_with()
>>> mock.__exit__.assert_called_with(None, None, None)
```

Calls to magic methods do not appear in *method_calls*, but they are recorded in *mock_calls*.

> **Note**
>
> If you use the *spec* keyword argument to create a mock then attempting to set a magic method that isn't in the spec will raise an *AttributeError*.

The full list of supported magic methods is :

- `__hash__`, `__sizeof__`, `__repr__` and `__str__`
- `__dir__`, `__format__` et `__subclasses__`
- `__round__`, `__floor__`, `__trunc__` and `__ceil__`
- Comparisons : `__lt__`, `__gt__`, `__le__`, `__ge__`, `__eq__` and `__ne__`
- Container methods : `__getitem__`, `__setitem__`, `__delitem__`, `__contains__`, `__len__`, `__iter__`, `__reversed__` and `__missing__`
- Context manager : `__enter__`, `__exit__`, `__aenter__` and `__aexit__`
- Unary numeric methods : `__neg__`, `__pos__` and `__invert__`
- The numeric methods (including right hand and in-place variants) : `__add__`, `__sub__`, `__mul__`, `__matmul__`, `__truediv__`, `__floordiv__`, `__mod__`, `__divmod__`, `__lshift__`, `__rshift__`, `__and__`, `__xor__`, `__or__`, and `__pow__`
- Numeric conversion methods : `__complex__`, `__int__`, `__float__` and `__index__`
- Descriptor methods : `__get__`, `__set__` and `__delete__`
- Pickling : `__reduce__`, `__reduce_ex__`, `__getinitargs__`, `__getnewargs__`, `__getstate__` and `__setstate__`
- File system path representation : `__fspath__`
- Asynchronous iteration methods : `__aiter__` and `__anext__`

Modifié dans la version 3.8 : Added support for *os.PathLike.__fspath__()*.

Modifié dans la version 3.8 : Added support for `__aenter__`, `__aexit__`, `__aiter__` and `__anext__`.

The following methods exist but are *not* supported as they are either in use by mock, can't be set dynamically, or can cause problems :

- `__getattr__`, `__setattr__`, `__init__` and `__new__`
- `__prepare__`, `__instancecheck__`, `__subclasscheck__`, `__del__`

Magic Mock

There are two `MagicMock` variants : *MagicMock* and *NonCallableMagicMock*.

class `unittest.mock.`**`MagicMock`**(**args*, ***kw*)

`MagicMock` is a subclass of *Mock* with default implementations of most of the *magic methods*. You can use `MagicMock` without having to configure the magic methods yourself.

The constructor parameters have the same meaning as for *Mock*.

If you use the *spec* or *spec_set* arguments then *only* magic methods that exist in the spec will be created.

class unittest.mock.**NonCallableMagicMock**(**args*, ***kw*)

A non-callable version of `MagicMock`.

The constructor parameters have the same meaning as for `MagicMock`, with the exception of *return_value* and *side_effect* which have no meaning on a non-callable mock.

The magic methods are setup with `MagicMock` objects, so you can configure them and use them in the usual way :

```
>>> mock = MagicMock()
>>> mock[3] = 'fish'
>>> mock.__setitem__.assert_called_with(3, 'fish')
>>> mock.__getitem__.return_value = 'result'
>>> mock[2]
'result'
```

By default many of the protocol methods are required to return objects of a specific type. These methods are preconfigured with a default return value, so that they can be used without you having to do anything if you aren't interested in the return value. You can still *set* the return value manually if you want to change the default.

Methods and their defaults :

- `__lt__` : `NotImplemented`
- `__gt__` : `NotImplemented`
- `__le__` : `NotImplemented`
- `__ge__` : `NotImplemented`
- `__int__` : `1`
- `__contains__` : `False`
- `__int__` : `0`
- `__iter__` : `iter([])`
- `__exit__` : `False`
- `__exit__` : `False`
- `__complex__` : `1j`
- `__float__` : `1.0`
- `__bool__` : `True`
- `__index__` : `1`
- `__hash__` : default hash for the mock
- `__str__` : default str for the mock
- `__sizeof__` : default sizeof for the mock

Par exemple :

```
>>> mock = MagicMock()
>>> int(mock)
1
>>> len(mock)
0
>>> list(mock)
[]
>>> object() in mock
False
```

The two equality methods, `__eq__()` and `__ne__()`, are special. They do the default equality comparison on identity, using the `side_effect` attribute, unless you change their return value to return something else :

```
>>> MagicMock() == 3
False
>>> MagicMock() != 3
True
>>> mock = MagicMock()
>>> mock.__eq__.return_value = True
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> mock == 3
True
```

The return value of `MagicMock.__iter__()` can be any iterable object and isn't required to be an iterator :

```
>>> mock = MagicMock()
>>> mock.__iter__.return_value = ['a', 'b', 'c']
>>> list(mock)
['a', 'b', 'c']
>>> list(mock)
['a', 'b', 'c']
```

If the return value *is* an iterator, then iterating over it once will consume it and subsequent iterations will result in an empty list :

```
>>> mock.__iter__.return_value = iter(['a', 'b', 'c'])
>>> list(mock)
['a', 'b', 'c']
>>> list(mock)
[]
```

`MagicMock` has all of the supported magic methods configured except for some of the obscure and obsolete ones. You can still set these up if you want.

Magic methods that are supported but not setup by default in `MagicMock` are :

- `__subclasses__`
- `__dir__`
- `__format__`
- `__get__`, `__set__` et `__delete__`
- `__reversed__` et `__missing__`
- `__reduce__`, `__reduce_ex__`, `__getinitargs__`, `__getnewargs__`, `__getstate__` and `__setstate__`
- `__getformat__`

27.6.5 Helpers

sentinel

`unittest.mock.`**`sentinel`**

The `sentinel` object provides a convenient way of providing unique objects for your tests.

Attributes are created on demand when you access them by name. Accessing the same attribute will always return the same object. The objects returned have a sensible repr so that test failure messages are readable.

Modifié dans la version 3.7 : The `sentinel` attributes now preserve their identity when they are *copied* or *pickled*.

Sometimes when testing you need to test that a specific object is passed as an argument to another method, or returned. It can be common to create named sentinel objects to test this. *sentinel* provides a convenient way of creating and testing the identity of objects like this.

In this example we monkey patch `method` to return `sentinel.some_object` :

```
>>> real = ProductionClass()
>>> real.method = Mock(name="method")
>>> real.method.return_value = sentinel.some_object
>>> result = real.method()
>>> assert result is sentinel.some_object
>>> result
sentinel.some_object
```

DEFAULT

unittest.mock.**DEFAULT**

The *DEFAULT* object is a pre-created sentinel (actually sentinel.DEFAULT). It can be used by *side_effect* functions to indicate that the normal return value should be used.

appel

unittest.mock.**call**(**args*, ***kwargs*)

call() is a helper object for making simpler assertions, for comparing with *call_args*, *call_args_list*, *mock_calls* and *method_calls*. *call()* can also be used with *assert_has_calls()*.

```
>>> m = MagicMock(return_value=None)
>>> m(1, 2, a='foo', b='bar')
>>> m()
>>> m.call_args_list == [call(1, 2, a='foo', b='bar'), call()]
True
```

call.**call_list**()

For a call object that represents multiple calls, *call_list()* returns a list of all the intermediate calls as well as the final call.

call_list is particularly useful for making assertions on "chained calls". A chained call is multiple calls on a single line of code. This results in multiple entries in *mock_calls* on a mock. Manually constructing the sequence of calls can be tedious.

call_list() can construct the sequence of calls from the same chained call :

```
>>> m = MagicMock()
>>> m(1).method(arg='foo').other('bar')(2.0)
<MagicMock name='mock().method().other()()' id='...'>
>>> kall = call(1).method(arg='foo').other('bar')(2.0)
>>> kall.call_list()
[call(1),
 call().method(arg='foo'),
 call().method().other('bar'),
 call().method().other()(2.0)]
>>> m.mock_calls == kall.call_list()
True
```

A call object is either a tuple of (positional args, keyword args) or (name, positional args, keyword args) depending on how it was constructed. When you construct them yourself this isn't particularly interesting, but the call objects that are in the *Mock.call_args*, *Mock.call_args_list* and *Mock.mock_calls* attributes can be introspected to get at the individual arguments they contain.

The call objects in *Mock.call_args* and *Mock.call_args_list* are two-tuples of (positional args, keyword args) whereas the call objects in *Mock.mock_calls*, along with ones you construct yourself, are three-tuples of (name, positional args, keyword args).

You can use their "tupleness" to pull out the individual arguments for more complex introspection and assertions. The positional arguments are a tuple (an empty tuple if there are no positional arguments) and the keyword arguments are a dictionary :

```
>>> m = MagicMock(return_value=None)
>>> m(1, 2, 3, arg='one', arg2='two')
>>> kall = m.call_args
>>> kall.args
(1, 2, 3)
>>> kall.kwargs
{'arg': 'one', 'arg2': 'two'}
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> kall.args is kall[0]
True
>>> kall.kwargs is kall[1]
True
```

```
>>> m = MagicMock()
>>> m.foo(4, 5, 6, arg='two', arg2='three')
<MagicMock name='mock.foo()' id='...'>
>>> kall = m.mock_calls[0]
>>> name, args, kwargs = kall
>>> name
'foo'
>>> args
(4, 5, 6)
>>> kwargs
{'arg': 'two', 'arg2': 'three'}
>>> name is m.mock_calls[0][0]
True
```

create_autospec

`unittest.mock.`**`create_autospec`**(*spec*, *spec_set=False*, *instance=False*, ***kwargs*)

Create a mock object using another object as a spec. Attributes on the mock will use the corresponding attribute on the *spec* object as their spec.

Functions or methods being mocked will have their arguments checked to ensure that they are called with the correct signature.

If *spec_set* is `True` then attempting to set attributes that don't exist on the spec object will raise an `AttributeError`.

If a class is used as a spec then the return value of the mock (the instance of the class) will have the same spec. You can use a class as the spec for an instance object by passing `instance=True`. The returned mock will only be callable if instances of the mock are callable.

`create_autospec()` also takes arbitrary keyword arguments that are passed to the constructor of the created mock.

See *Autospeccing* for examples of how to use auto-speccing with `create_autospec()` and the *autospec* argument to `patch()`.

Modifié dans la version 3.8 : `create_autospec()` now returns an `AsyncMock` if the target is an async function.

ANY

`unittest.mock.`**`ANY`**

Sometimes you may need to make assertions about *some* of the arguments in a call to mock, but either not care about some of the arguments or want to pull them individually out of `call_args` and make more complex assertions on them.

To ignore certain arguments you can pass in objects that compare equal to *everything*. Calls to `assert_called_with()` and `assert_called_once_with()` will then succeed no matter what was passed in.

```
>>> mock = Mock(return_value=None)
>>> mock('foo', bar=object())
>>> mock.assert_called_once_with('foo', bar=ANY)
```

`ANY` can also be used in comparisons with call lists like `mock_calls` :

```
>>> m = MagicMock(return_value=None)
>>> m(1)
>>> m(1, 2)
>>> m(object())
>>> m.mock_calls == [call(1), call(1, 2), ANY]
True
```

ANY is not limited to comparisons with call objects and so can also be used in test assertions :

```
class TestStringMethods(unittest.TestCase):

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', ANY])
```

FILTER_DIR

unittest.mock.**FILTER_DIR**

FILTER_DIR is a module level variable that controls the way mock objects respond to *dir()*. The default is `True`, which uses the filtering described below, to only show useful members. If you dislike this filtering, or need to switch it off for diagnostic purposes, then set `mock.FILTER_DIR = False`.

With filtering on, `dir(some_mock)` shows only useful attributes and will include any dynamically created attributes that wouldn't normally be shown. If the mock was created with a *spec* (or *autospec* of course) then all the attributes from the original are shown, even if they haven't been accessed yet :

```
>>> dir(Mock())
['assert_any_call',
 'assert_called',
 'assert_called_once',
 'assert_called_once_with',
 'assert_called_with',
 'assert_has_calls',
 'assert_not_called',
 'attach_mock',
 ...
>>> from urllib import request
>>> dir(Mock(spec=request))
['AbstractBasicAuthHandler',
 'AbstractDigestAuthHandler',
 'AbstractHTTPHandler',
 'BaseHandler',
 ...
```

Many of the not-very-useful (private to *Mock* rather than the thing being mocked) underscore and double underscore prefixed attributes have been filtered from the result of calling *dir()* on a *Mock*. If you dislike this behaviour you can switch it off by setting the module level switch *FILTER_DIR* :

```
>>> from unittest import mock
>>> mock.FILTER_DIR = False
>>> dir(mock.Mock())
['_NonCallableMock__get_return_value',
 '_NonCallableMock__get_side_effect',
 '_NonCallableMock__return_value_doc',
 '_NonCallableMock__set_return_value',
 '_NonCallableMock__set_side_effect',
 '__call__',
```

(suite sur la page suivante)

(suite de la page précédente)

```
 '__class__',
 ...
```

Alternatively you can just use `vars(my_mock)` (instance members) and `dir(type(my_mock))` (type members) to bypass the filtering irrespective of `FILTER_DIR`.

mock_open

`unittest.mock.`**`mock_open`**(*mock=None*, *read_data=None*)

A helper function to create a mock to replace the use of `open()`. It works for `open()` called directly or used as a context manager.

The *mock* argument is the mock object to configure. If `None` (the default) then a `MagicMock` will be created for you, with the API limited to methods or attributes available on standard file handles.

read_data is a string for the `read()`, `readline()`, and `readlines()` methods of the file handle to return. Calls to those methods will take data from *read_data* until it is depleted. The mock of these methods is pretty simplistic : every time the *mock* is called, the *read_data* is rewound to the start. If you need more control over the data that you are feeding to the tested code you will need to customize this mock for yourself. When that is insufficient, one of the in-memory filesystem packages on PyPI can offer a realistic filesystem for testing.

Modifié dans la version 3.4 : Added `readline()` and `readlines()` support. The mock of `read()` changed to consume *read_data* rather than returning it on each call.

Modifié dans la version 3.5 : *read_data* is now reset on each call to the *mock*.

Modifié dans la version 3.8 : Added `__iter__()` to implementation so that iteration (such as in for loops) correctly consumes *read_data*.

Using `open()` as a context manager is a great way to ensure your file handles are closed properly and is becoming common :

```
with open('/some/path', 'w') as f:
    f.write('something')
```

The issue is that even if you mock out the call to `open()` it is the *returned object* that is used as a context manager (and has `__enter__()` and `__exit__()` called).

Mocking context managers with a `MagicMock` is common enough and fiddly enough that a helper function is useful.

```
>>> m = mock_open()
>>> with patch('__main__.open', m):
...     with open('foo', 'w') as h:
...         h.write('some stuff')
...
>>> m.mock_calls
[call('foo', 'w'),
 call().__enter__(),
 call().write('some stuff'),
 call().__exit__(None, None, None)]
>>> m.assert_called_once_with('foo', 'w')
>>> handle = m()
>>> handle.write.assert_called_once_with('some stuff')
```

And for reading files :

```
>>> with patch('__main__.open', mock_open(read_data='bibble')) as m:
...     with open('foo') as h:
...         result = h.read()
...
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> m.assert_called_once_with('foo')
>>> assert result == 'bibble'
```

Autospeccing

Autospeccing is based on the existing `spec` feature of mock. It limits the api of mocks to the api of an original object (the spec), but it is recursive (implemented lazily) so that attributes of mocks only have the same api as the attributes of the spec. In addition mocked functions / methods have the same call signature as the original so they raise a *TypeError* if they are called incorrectly.

Before I explain how auto-speccing works, here's why it is needed.

Mock is a very powerful and flexible object, but it suffers from a flaw which is general to mocking. If you refactor some of your code, rename members and so on, any tests for code that is still using the *old api* but uses mocks instead of the real objects will still pass. This means your tests can all pass even though your code is broken.

Modifié dans la version 3.5 : Before 3.5, tests with a typo in the word assert would silently pass when they should raise an error. You can still achieve this behavior by passing `unsafe=True` to Mock.

Note that this is another reason why you need integration tests as well as unit tests. Testing everything in isolation is all fine and dandy, but if you don't test how your units are "wired together" there is still lots of room for bugs that tests might have caught.

unittest.mock already provides a feature to help with this, called speccing. If you use a class or instance as the `spec` for a mock then you can only access attributes on the mock that exist on the real class :

```
>>> from urllib import request
>>> mock = Mock(spec=request.Request)
>>> mock.assret_called_with  # Intentional typo!
Traceback (most recent call last):
 ...
AttributeError: Mock object has no attribute 'assret_called_with'
```

The spec only applies to the mock itself, so we still have the same issue with any methods on the mock :

```
>>> mock.has_data()
<mock.Mock object at 0x...>
>>> mock.has_data.assret_called_with()  # Intentional typo!
```

Auto-speccing solves this problem. You can either pass `autospec=True` to *patch()* / *patch.object()* or use the *create_autospec()* function to create a mock with a spec. If you use the `autospec=True` argument to *patch()* then the object that is being replaced will be used as the spec object. Because the speccing is done "lazily" (the spec is created as attributes on the mock are accessed) you can use it with very complex or deeply nested objects (like modules that import modules that import modules) without a big performance hit.

Here's an example of it in use :

```
>>> from urllib import request
>>> patcher = patch('__main__.request', autospec=True)
>>> mock_request = patcher.start()
>>> request is mock_request
True
>>> mock_request.Request
<MagicMock name='request.Request' spec='Request' id='...'>
```

You can see that `request.Request` has a spec. `request.Request` takes two arguments in the constructor (one of which is *self*). Here's what happens if we try to call it incorrectly :

```
>>> req = request.Request()
Traceback (most recent call last):
```

(suite sur la page suivante)

(suite de la page précédente)

```
 ...
TypeError: <lambda>() takes at least 2 arguments (1 given)
```

The spec also applies to instantiated classes (i.e. the return value of specced mocks) :

```
>>> req = request.Request('foo')
>>> req
<NonCallableMagicMock name='request.Request()' spec='Request' id='...'>
```

`Request` objects are not callable, so the return value of instantiating our mocked out `request.Request` is a non-callable mock. With the spec in place any typos in our asserts will raise the correct error :

```
>>> req.add_header('spam', 'eggs')
<MagicMock name='request.Request().add_header()' id='...'>
>>> req.add_header.assret_called_with  # Intentional typo!
Traceback (most recent call last):
 ...
AttributeError: Mock object has no attribute 'assret_called_with'
>>> req.add_header.assert_called_with('spam', 'eggs')
```

In many cases you will just be able to add `autospec=True` to your existing *patch()* calls and then be protected against bugs due to typos and api changes.

As well as using *autospec* through *patch()* there is a *create_autospec()* for creating autospecced mocks directly :

```
>>> from urllib import request
>>> mock_request = create_autospec(request)
>>> mock_request.Request('foo', 'bar')
<NonCallableMagicMock name='mock.Request()' spec='Request' id='...'>
```

This isn't without caveats and limitations however, which is why it is not the default behaviour. In order to know what attributes are available on the spec object, autospec has to introspect (access attributes) the spec. As you traverse attributes on the mock a corresponding traversal of the original object is happening under the hood. If any of your specced objects have properties or descriptors that can trigger code execution then you may not be able to use autospec. On the other hand it is much better to design your objects so that introspection is safe [4].

A more serious problem is that it is common for instance attributes to be created in the `__init__()` method and not to exist on the class at all. *autospec* can't know about any dynamically created attributes and restricts the api to visible attributes.

```
>>> class Something:
...   def __init__(self):
...     self.a = 33
...
>>> with patch('__main__.Something', autospec=True):
...   thing = Something()
...   thing.a
...
Traceback (most recent call last):
  ...
AttributeError: Mock object has no attribute 'a'
```

There are a few different ways of resolving this problem. The easiest, but not necessarily the least annoying, way is to simply set the required attributes on the mock after creation. Just because *autospec* doesn't allow you to fetch attributes that don't exist on the spec it doesn't prevent you setting them :

4. This only applies to classes or already instantiated objects. Calling a mocked class to create a mock instance *does not* create a real instance. It is only attribute lookups - along with calls to *dir()* - that are done.

```
>>> with patch('__main__.Something', autospec=True):
...   thing = Something()
...   thing.a = 33
...
```

There is a more aggressive version of both *spec* and *autospec* that *does* prevent you setting non-existent attributes. This is useful if you want to ensure your code only *sets* valid attributes too, but obviously it prevents this particular scenario :

```
>>> with patch('__main__.Something', autospec=True, spec_set=True):
...   thing = Something()
...   thing.a = 33
...
Traceback (most recent call last):
 ...
AttributeError: Mock object has no attribute 'a'
```

Probably the best way of solving the problem is to add class attributes as default values for instance members initialised in `__init__()`. Note that if you are only setting default attributes in `__init__()` then providing them via class attributes (shared between instances of course) is faster too. e.g.

```
class Something:
    a = 33
```

This brings up another issue. It is relatively common to provide a default value of `None` for members that will later be an object of a different type. `None` would be useless as a spec because it wouldn't let you access *any* attributes or methods on it. As `None` is *never* going to be useful as a spec, and probably indicates a member that will normally of some other type, autospec doesn't use a spec for members that are set to `None`. These will just be ordinary mocks (well - MagicMocks) :

```
>>> class Something:
...     member = None
...
>>> mock = create_autospec(Something)
>>> mock.member.foo.bar.baz()
<MagicMock name='mock.member.foo.bar.baz()' id='...'>
```

If modifying your production classes to add defaults isn't to your liking then there are more options. One of these is simply to use an instance as the spec rather than the class. The other is to create a subclass of the production class and add the defaults to the subclass without affecting the production class. Both of these require you to use an alternative object as the spec. Thankfully *`patch()`* supports this - you can simply pass the alternative object as the *autospec* argument :

```
>>> class Something:
...   def __init__(self):
...     self.a = 33
...
>>> class SomethingForTest(Something):
...   a = 33
...
>>> p = patch('__main__.Something', autospec=SomethingForTest)
>>> mock = p.start()
>>> mock.a
<NonCallableMagicMock name='Something.a' spec='int' id='...'>
```

Sealing mocks

`unittest.mock.seal(mock)`

Seal will disable the automatic creation of mocks when accessing an attribute of the mock being sealed or any of its attributes that are already mocks recursively.

If a mock instance with a name or a spec is assigned to an attribute it won't be considered in the sealing chain. This allows one to prevent seal from fixing part of the mock object.

```
>>> mock = Mock()
>>> mock.submock.attribute1 = 2
>>> mock.not_submock = mock.Mock(name="sample_name")
>>> seal(mock)
>>> mock.new_attribute  # This will raise AttributeError.
>>> mock.submock.attribute2  # This will raise AttributeError.
>>> mock.not_submock.attribute2  # This won't raise.
```

Ajouté dans la version 3.7.

27.6.6 Order of precedence of `side_effect`, `return_value` and *wraps*

The order of their precedence is :

1. `side_effect`
2. `return_value`
3. *wraps*

If all three are set, mock will return the value from `side_effect`, ignoring `return_value` and the wrapped object altogether. If any two are set, the one with the higher precedence will return the value. Regardless of the order of which was set first, the order of precedence remains unchanged.

```
>>> from unittest.mock import Mock
>>> class Order:
...     @staticmethod
...     def get_value():
...         return "third"
...
>>> order_mock = Mock(spec=Order, wraps=Order)
>>> order_mock.get_value.side_effect = ["first"]
>>> order_mock.get_value.return_value = "second"
>>> order_mock.get_value()
'first'
```

As `None` is the default value of `side_effect`, if you reassign its value back to `None`, the order of precedence will be checked between `return_value` and the wrapped object, ignoring `side_effect`.

```
>>> order_mock.get_value.side_effect = None
>>> order_mock.get_value()
'second'
```

If the value being returned by `side_effect` is `DEFAULT`, it is ignored and the order of precedence moves to the successor to obtain the value to return.

```
>>> from unittest.mock import DEFAULT
>>> order_mock.get_value.side_effect = [DEFAULT]
>>> order_mock.get_value()
'second'
```

When `Mock` wraps an object, the default value of `return_value` will be `DEFAULT`.

```
>>> order_mock = Mock(spec=Order, wraps=Order)
>>> order_mock.return_value
sentinel.DEFAULT
>>> order_mock.get_value.return_value
sentinel.DEFAULT
```

The order of precedence will ignore this value and it will move to the last successor which is the wrapped object.

As the real call is being made to the wrapped object, creating an instance of this mock will return the real instance of the class. The positional arguments, if any, required by the wrapped object must be passed.

```
>>> order_mock_instance = order_mock()
>>> isinstance(order_mock_instance, Order)
True
>>> order_mock_instance.get_value()
'third'
```

```
>>> order_mock.get_value.return_value = DEFAULT
>>> order_mock.get_value()
'third'
```

```
>>> order_mock.get_value.return_value = "second"
>>> order_mock.get_value()
'second'
```

But if you assign `None` to it, this will not be ignored as it is an explicit assignment. So, the order of precedence will not move to the wrapped object.

```
>>> order_mock.get_value.return_value = None
>>> order_mock.get_value() is None
True
```

Even if you set all three at once when initializing the mock, the order of precedence remains the same :

```
>>> order_mock = Mock(spec=Order, wraps=Order,
...                   **{"get_value.side_effect": ["first"],
...                      "get_value.return_value": "second"}
...                   )
...
>>> order_mock.get_value()
'first'
>>> order_mock.get_value.side_effect = None
>>> order_mock.get_value()
'second'
>>> order_mock.get_value.return_value = DEFAULT
>>> order_mock.get_value()
'third'
```

If *side_effect* is exhausted, the order of precedence will not cause a value to be obtained from the successors. Instead, `StopIteration` exception is raised.

```
>>> order_mock = Mock(spec=Order, wraps=Order)
>>> order_mock.get_value.side_effect = ["first side effect value",
...                                     "another side effect value"]
>>> order_mock.get_value.return_value = "second"
```

```
>>> order_mock.get_value()
'first side effect value'
>>> order_mock.get_value()
'another side effect value'
```

```
>>> order_mock.get_value()
Traceback (most recent call last):
 ...
StopIteration
```

27.7 unittest.mock --- getting started

Ajouté dans la version 3.3.

27.7.1 Utilisation de Mock ou l'art de singer

Simulation des méthodes

Usages courant de *Mock* :

— Substitution des méthodes
— Enregistrement des appels faits sur les objets

On peut remplacer une méthode sur un objet pour contrôler qu'elle est bien appelée avec le nombre correct d'arguments :

```
>>> real = SomeClass()
>>> real.method = MagicMock(name='method')
>>> real.method(3, 4, 5, key='value')
<MagicMock name='method()' id='...'>
```

Une fois notre objet simulacre appelé (via `real.method` dans notre exemple), il fournit des méthodes et attributs permettant de valider comment il a été appelé.

> **Note**
>
> Dans la majeure partie des exemples donnés ici, les classes *Mock* et *MagicMock* sont interchangeables. Étant donné que `MagicMock` est la classe la plus puissante des deux, cela fait sens de l'utiliser par défaut.

Une fois l'objet Mock appelé, son attribut *called* est défini à `True`. Qui plus est, nous pouvons utiliser les méthodes *assert_called_with()* ou *assert_called_once_with()* pour contrôler qu'il a été appelé avec les bons arguments.

Cet exemple teste que l'appel de la méthode `ProductionClass().method` implique bien celui de la méthode `something` :

```
>>> class ProductionClass:
...     def method(self):
...         self.something(1, 2, 3)
...     def something(self, a, b, c):
...         pass
...
>>> real = ProductionClass()
>>> real.something = MagicMock()
>>> real.method()
>>> real.something.assert_called_once_with(1, 2, 3)
```

S'assurer de la bonne utilisation d'un objet

Dans l'exemple précédent, nous avons directement remplacé une méthode par un objet (afin de valider que l'appel était correct). Une autre façon de faire est de passer un objet Mock en argument d'une méthode (ou de tout autre partie du code à tester) et ensuite de contrôler que notre objet a été utilisé de la façon attendue.

Ci-dessous, `ProductionClass` dispose d'une méthode `closer`. Si on l'appelle avec un objet, alors elle appelle la méthode `close` dessus.

```
>>> class ProductionClass:
...     def closer(self, something):
...         something.close()
...
```

Ainsi, pour tester cette classe, nous devons lui passer un objet ayant une méthode `close`, puis vérifier qu'elle a bien été appelée.

```
>>> real = ProductionClass()
>>> mock = Mock()
>>> real.closer(mock)
>>> mock.close.assert_called_with()
```

En fait, nous n'avons pas à nous soucier de fournir la méthode `close` dans notre objet « simulé ». Le simple fait d'accéder à la méthode `close` l'a crée. Si par contre la méthode `close` n'a pas été appelée alors, bien que le test la créée en y accédant, *`assert_called_with()`* lèvera une exception.

Simulation des classes

Un cas d'utilisation courant consiste à émuler les classes instanciées par le code que nous testons. Quand on *patch* une classe, alors cette classe est remplacée par un objet *mock*. Les instances de la classe étant créées en *appelant la classe*, on accède à « l'instance *mock* » via la valeur de retour de la classe émulée.

In the example below we have a function `some_function` that instantiates `Foo` and calls a method on it. The call to *`patch()`* replaces the class `Foo` with a mock. The `Foo` instance is the result of calling the mock, so it is configured by modifying the mock *`return_value`*.

```
>>> def some_function():
...     instance = module.Foo()
...     return instance.method()
...
>>> with patch('module.Foo') as mock:
...     instance = mock.return_value
...     instance.method.return_value = 'the result'
...     result = some_function()
...     assert result == 'the result'
```

Naming your mocks

It can be useful to give your mocks a name. The name is shown in the repr of the mock and can be helpful when the mock appears in test failure messages. The name is also propagated to attributes or methods of the mock :

```
>>> mock = MagicMock(name='foo')
>>> mock
<MagicMock name='foo' id='...'>
>>> mock.method
<MagicMock name='foo.method' id='...'>
```

Tracking all Calls

Often you want to track more than a single call to a method. The *mock_calls* attribute records all calls to child attributes of the mock - and also to their children.

```
>>> mock = MagicMock()
>>> mock.method()
<MagicMock name='mock.method()' id='...'>
>>> mock.attribute.method(10, x=53)
<MagicMock name='mock.attribute.method()' id='...'>
>>> mock.mock_calls
[call.method(), call.attribute.method(10, x=53)]
```

If you make an assertion about `mock_calls` and any unexpected methods have been called, then the assertion will fail. This is useful because as well as asserting that the calls you expected have been made, you are also checking that they were made in the right order and with no additional calls :

You use the *call* object to construct lists for comparing with `mock_calls` :

```
>>> expected = [call.method(), call.attribute.method(10, x=53)]
>>> mock.mock_calls == expected
True
```

However, parameters to calls that return mocks are not recorded, which means it is not possible to track nested calls where the parameters used to create ancestors are important :

```
>>> m = Mock()
>>> m.factory(important=True).deliver()
<Mock name='mock.factory().deliver()' id='...'>
>>> m.mock_calls[-1] == call.factory(important=False).deliver()
True
```

Setting Return Values and Attributes

Setting the return values on a mock object is trivially easy :

```
>>> mock = Mock()
>>> mock.return_value = 3
>>> mock()
3
```

Of course you can do the same for methods on the mock :

```
>>> mock = Mock()
>>> mock.method.return_value = 3
>>> mock.method()
3
```

The return value can also be set in the constructor :

```
>>> mock = Mock(return_value=3)
>>> mock()
3
```

If you need an attribute setting on your mock, just do it :

```
>>> mock = Mock()
>>> mock.x = 3
>>> mock.x
3
```

Sometimes you want to mock up a more complex situation, like for example `mock.connection.cursor().execute("SELECT 1")`. If we wanted this call to return a list, then we have to configure the result of the nested call.

We can use *call* to construct the set of calls in a "chained call" like this for easy assertion afterwards :

```
>>> mock = Mock()
>>> cursor = mock.connection.cursor.return_value
>>> cursor.execute.return_value = ['foo']
>>> mock.connection.cursor().execute("SELECT 1")
['foo']
>>> expected = call.connection.cursor().execute("SELECT 1").call_list()
>>> mock.mock_calls
[call.connection.cursor(), call.connection.cursor().execute('SELECT 1')]
>>> mock.mock_calls == expected
True
```

It is the call to `.call_list()` that turns our call object into a list of calls representing the chained calls.

Raising exceptions with mocks

A useful attribute is *side_effect*. If you set this to an exception class or instance then the exception will be raised when the mock is called.

```
>>> mock = Mock(side_effect=Exception('Boom!'))
>>> mock()
Traceback (most recent call last):
  ...
Exception: Boom!
```

Side effect functions and iterables

`side_effect` can also be set to a function or an iterable. The use case for `side_effect` as an iterable is where your mock is going to be called several times, and you want each call to return a different value. When you set `side_effect` to an iterable every call to the mock returns the next value from the iterable :

```
>>> mock = MagicMock(side_effect=[4, 5, 6])
>>> mock()
4
>>> mock()
5
>>> mock()
6
```

For more advanced use cases, like dynamically varying the return values depending on what the mock is called with, `side_effect` can be a function. The function will be called with the same arguments as the mock. Whatever the function returns is what the call returns :

```
>>> vals = {(1, 2): 1, (2, 3): 2}
>>> def side_effect(*args):
...     return vals[args]
...
>>> mock = MagicMock(side_effect=side_effect)
>>> mock(1, 2)
1
>>> mock(2, 3)
2
```

Mocking asynchronous iterators

Since Python 3.8, `AsyncMock` and `MagicMock` have support to mock async-iterators through `__aiter__`. The `return_value` attribute of `__aiter__` can be used to set the return values to be used for iteration.

```
>>> mock = MagicMock()  # AsyncMock also works here
>>> mock.__aiter__.return_value = [1, 2, 3]
>>> async def main():
...     return [i async for i in mock]
...
>>> asyncio.run(main())
[1, 2, 3]
```

Mocking asynchronous context manager

Since Python 3.8, `AsyncMock` and `MagicMock` have support to mock async-context-managers through `__aenter__` and `__aexit__`. By default, `__aenter__` and `__aexit__` are `AsyncMock` instances that return an async function.

```
>>> class AsyncContextManager:
...     async def __aenter__(self):
...         return self
...     async def __aexit__(self, exc_type, exc, tb):
...         pass
...
>>> mock_instance = MagicMock(AsyncContextManager())  # AsyncMock also works here
>>> async def main():
...     async with mock_instance as result:
...         pass
...
>>> asyncio.run(main())
>>> mock_instance.__aenter__.assert_awaited_once()
>>> mock_instance.__aexit__.assert_awaited_once()
```

Creating a Mock from an Existing Object

One problem with over use of mocking is that it couples your tests to the implementation of your mocks rather than your real code. Suppose you have a class that implements `some_method`. In a test for another class, you provide a mock of this object that *also* provides `some_method`. If later you refactor the first class, so that it no longer has `some_method` - then your tests will continue to pass even though your code is now broken !

`Mock` allows you to provide an object as a specification for the mock, using the *spec* keyword argument. Accessing methods / attributes on the mock that don't exist on your specification object will immediately raise an attribute error. If you change the implementation of your specification, then tests that use that class will start failing immediately without you having to instantiate the class in those tests.

```
>>> mock = Mock(spec=SomeClass)
>>> mock.old_method()
Traceback (most recent call last):
   ...
AttributeError: Mock object has no attribute 'old_method'. Did you mean: 'class_
↪method'?
```

Using a specification also enables a smarter matching of calls made to the mock, regardless of whether some parameters were passed as positional or named arguments :

```
>>> def f(a, b, c): pass
...
>>> mock = Mock(spec=f)
>>> mock(1, 2, 3)
```

(suite sur la page suivante)

(suite de la page précédente)

```
<Mock name='mock()' id='140161580456576'>
>>> mock.assert_called_with(a=1, b=2, c=3)
```

If you want this smarter matching to also work with method calls on the mock, you can use *auto-speccing*.

If you want a stronger form of specification that prevents the setting of arbitrary attributes as well as the getting of them then you can use *spec_set* instead of *spec*.

Using side_effect to return per file content

`mock_open()` is used to patch `open()` method. `side_effect` can be used to return a new Mock object per call. This can be used to return different contents per file stored in a dictionary :

```
DEFAULT = "default"
data_dict = {"file1": "data1",
             "file2": "data2"}

def open_side_effect(name):
    return mock_open(read_data=data_dict.get(name, DEFAULT))()

with patch("builtins.open", side_effect=open_side_effect):
    with open("file1") as file1:
        assert file1.read() == "data1"

    with open("file2") as file2:
        assert file2.read() == "data2"

    with open("file3") as file2:
        assert file2.read() == "default"
```

27.7.2 Patch Decorators

Note

Avec `patch()`, il est important de *patcher* les objets dans l'espace de nommage où ils sont recherchés. C'est ce qui se fait normalement, mais pour un guide rapide, lisez *où patcher*.

A common need in tests is to patch a class attribute or a module attribute, for example patching a builtin or patching a class in a module to test that it is instantiated. Modules and classes are effectively global, so patching on them has to be undone after the test or the patch will persist into other tests and cause hard to diagnose problems.

mock provides three convenient decorators for this : `patch()`, `patch.object()` and `patch.dict()`. `patch` takes a single string, of the form `package.module.Class.attribute` to specify the attribute you are patching. It also optionally takes a value that you want the attribute (or class or whatever) to be replaced with. 'patch.object' takes an object and the name of the attribute you would like patched, plus optionally the value to patch it with.

`patch.object` :

```
>>> original = SomeClass.attribute
>>> @patch.object(SomeClass, 'attribute', sentinel.attribute)
... def test():
...     assert SomeClass.attribute == sentinel.attribute
...
>>> test()
>>> assert SomeClass.attribute == original
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> @patch('package.module.attribute', sentinel.attribute)
... def test():
...     from package.module import attribute
...     assert attribute is sentinel.attribute
...
>>> test()
```

If you are patching a module (including *builtins*) then use *patch()* instead of *patch.object()* :

```
>>> mock = MagicMock(return_value=sentinel.file_handle)
>>> with patch('builtins.open', mock):
...     handle = open('filename', 'r')
...
>>> mock.assert_called_with('filename', 'r')
>>> assert handle == sentinel.file_handle, "incorrect file handle returned"
```

The module name can be 'dotted', in the form `package.module` if needed :

```
>>> @patch('package.module.ClassName.attribute', sentinel.attribute)
... def test():
...     from package.module import ClassName
...     assert ClassName.attribute == sentinel.attribute
...
>>> test()
```

A nice pattern is to actually decorate test methods themselves :

```
>>> class MyTest(unittest.TestCase):
...     @patch.object(SomeClass, 'attribute', sentinel.attribute)
...     def test_something(self):
...         self.assertEqual(SomeClass.attribute, sentinel.attribute)
...
>>> original = SomeClass.attribute
>>> MyTest('test_something').test_something()
>>> assert SomeClass.attribute == original
```

If you want to patch with a Mock, you can use *patch()* with only one argument (or *patch.object()* with two arguments). The mock will be created for you and passed into the test function / method :

```
>>> class MyTest(unittest.TestCase):
...     @patch.object(SomeClass, 'static_method')
...     def test_something(self, mock_method):
...         SomeClass.static_method()
...         mock_method.assert_called_with()
...
>>> MyTest('test_something').test_something()
```

You can stack up multiple patch decorators using this pattern :

```
>>> class MyTest(unittest.TestCase):
...     @patch('package.module.ClassName1')
...     @patch('package.module.ClassName2')
...     def test_something(self, MockClass2, MockClass1):
...         self.assertIs(package.module.ClassName1, MockClass1)
...         self.assertIs(package.module.ClassName2, MockClass2)
...
>>> MyTest('test_something').test_something()
```

When you nest patch decorators the mocks are passed in to the decorated function in the same order they applied (the normal *Python* order that decorators are applied). This means from the bottom up, so in the example above the mock for `test_module.ClassName2` is passed in first.

Il existe également *patch.dict()* pour définir des valeurs d'un dictionnaire au sein d'une portée et restaurer ce dictionnaire à son état d'origine lorsque le test se termine :

```
>>> foo = {'key': 'value'}
>>> original = foo.copy()
>>> with patch.dict(foo, {'newkey': 'newvalue'}, clear=True):
...     assert foo == {'newkey': 'newvalue'}
...
>>> assert foo == original
```

`patch`, `patch.object` and `patch.dict` can all be used as context managers.

Where you use *patch()* to create a mock for you, you can get a reference to the mock using the "as" form of the with statement :

```
>>> class ProductionClass:
...     def method(self):
...         pass
...
>>> with patch.object(ProductionClass, 'method') as mock_method:
...     mock_method.return_value = None
...     real = ProductionClass()
...     real.method(1, 2, 3)
...
>>> mock_method.assert_called_with(1, 2, 3)
```

As an alternative `patch`, `patch.object` and `patch.dict` can be used as class decorators. When used in this way it is the same as applying the decorator individually to every method whose name starts with "test".

27.7.3 Further Examples

Here are some more examples for some slightly more advanced scenarios.

Mocking chained calls

Mocking chained calls is actually straightforward with mock once you understand the *return_value* attribute. When a mock is called for the first time, or you fetch its `return_value` before it has been called, a new *Mock* is created.

This means that you can see how the object returned from a call to a mocked object has been used by interrogating the `return_value` mock :

```
>>> mock = Mock()
>>> mock().foo(a=2, b=3)
<Mock name='mock().foo()' id='...'>
>>> mock.return_value.foo.assert_called_with(a=2, b=3)
```

From here it is a simple step to configure and then make assertions about chained calls. Of course another alternative is writing your code in a more testable way in the first place...

So, suppose we have some code that looks a little bit like this :

```
>>> class Something:
...     def __init__(self):
...         self.backend = BackendProvider()
...     def method(self):
```

(suite sur la page suivante)

(suite de la page précédente)

```
...         response = self.backend.get_endpoint('foobar').create_call('spam',
↪'eggs').start_call()
...         # more code
```

Assuming that `BackendProvider` is already well tested, how do we test `method()` ? Specifically, we want to test that the code section `# more code` uses the response object in the correct way.

As this chain of calls is made from an instance attribute we can monkey patch the `backend` attribute on a `Something` instance. In this particular case we are only interested in the return value from the final call to `start_call` so we don't have much configuration to do. Let's assume the object it returns is 'file-like', so we'll ensure that our response object uses the builtin *open()* as its `spec`.

To do this we create a mock instance as our mock backend and create a mock response object for it. To set the response as the return value for that final `start_call` we could do this :

```
mock_backend.get_endpoint.return_value.create_call.return_value.start_call.return_
↪value = mock_response
```

We can do that in a slightly nicer way using the *configure_mock()* method to directly set the return value for us :

```
>>> something = Something()
>>> mock_response = Mock(spec=open)
>>> mock_backend = Mock()
>>> config = {'get_endpoint.return_value.create_call.return_value.start_call.
↪return_value': mock_response}
>>> mock_backend.configure_mock(**config)
```

With these we monkey patch the "mock backend" in place and can make the real call :

```
>>> something.backend = mock_backend
>>> something.method()
```

Using *mock_calls* we can check the chained call with a single assert. A chained call is several calls in one line of code, so there will be several entries in `mock_calls`. We can use *call.call_list()* to create this list of calls for us :

```
>>> chained = call.get_endpoint('foobar').create_call('spam', 'eggs').start_call()
>>> call_list = chained.call_list()
>>> assert mock_backend.mock_calls == call_list
```

Partial mocking

In some tests I wanted to mock out a call to *datetime.date.today()* to return a known date, but I didn't want to prevent the code under test from creating new date objects. Unfortunately *datetime.date* is written in C, and so I couldn't just monkey-patch out the static *datetime.date.today()* method.

I found a simple way of doing this that involved effectively wrapping the date class with a mock, but passing through calls to the constructor to the real class (and returning real instances).

The *patch decorator* is used here to mock out the `date` class in the module under test. The *side_effect* attribute on the mock date class is then set to a lambda function that returns a real date. When the mock date class is called a real date will be constructed and returned by `side_effect`.

```
>>> from datetime import date
>>> with patch('mymodule.date') as mock_date:
...     mock_date.today.return_value = date(2010, 10, 8)
...     mock_date.side_effect = lambda *args, **kw: date(*args, **kw)
...
```

(suite sur la page suivante)

(suite de la page précédente)

```
...     assert mymodule.date.today() == date(2010, 10, 8)
...     assert mymodule.date(2009, 6, 8) == date(2009, 6, 8)
```

Note that we don't patch *datetime.date* globally, we patch `date` in the module that *uses* it. See *where to patch*.

When `date.today()` is called a known date is returned, but calls to the `date(...)` constructor still return normal dates. Without this you can find yourself having to calculate an expected result using exactly the same algorithm as the code under test, which is a classic testing anti-pattern.

Calls to the date constructor are recorded in the `mock_date` attributes (`call_count` and friends) which may also be useful for your tests.

An alternative way of dealing with mocking dates, or other builtin classes, is discussed in this blog entry.

Mocking a Generator Method

A Python generator is a function or method that uses the `yield` statement to return a series of values when iterated over [1].

A generator method / function is called to return the generator object. It is the generator object that is then iterated over. The protocol method for iteration is *__iter__()*, so we can mock this using a *MagicMock*.

Here's an example class with an "iter" method implemented as a generator :

```
>>> class Foo:
...     def iter(self):
...         for i in [1, 2, 3]:
...             yield i
...
>>> foo = Foo()
>>> list(foo.iter())
[1, 2, 3]
```

How would we mock this class, and in particular its "iter" method ?

To configure the values returned from the iteration (implicit in the call to *list*), we need to configure the object returned by the call to `foo.iter()`.

```
>>> mock_foo = MagicMock()
>>> mock_foo.iter.return_value = iter([1, 2, 3])
>>> list(mock_foo.iter())
[1, 2, 3]
```

Applying the same patch to every test method

If you want several patches in place for multiple test methods the obvious way is to apply the patch decorators to every method. This can feel like unnecessary repetition. Instead, you can use *patch()* (in all its various forms) as a class decorator. This applies the patches to all test methods on the class. A test method is identified by methods whose names start with `test` :

```
>>> @patch('mymodule.SomeClass')
... class MyTest(unittest.TestCase):
...
...     def test_one(self, MockSomeClass):
...         self.assertIs(mymodule.SomeClass, MockSomeClass)
...
...     def test_two(self, MockSomeClass):
```

(suite sur la page suivante)

1. There are also generator expressions and more advanced uses of generators, but we aren't concerned about them here. A very good introduction to generators and how powerful they are is : Generator Tricks for Systems Programmers.

(suite de la page précédente)

```
...         self.assertIs(mymodule.SomeClass, MockSomeClass)
...
...     def not_a_test(self):
...         return 'something'
...
>>> MyTest('test_one').test_one()
>>> MyTest('test_two').test_two()
>>> MyTest('test_two').not_a_test()
'something'
```

An alternative way of managing patches is to use the *patch methods : start and stop*. These allow you to move the patching into your `setUp` and `tearDown` methods.

```
>>> class MyTest(unittest.TestCase):
...     def setUp(self):
...         self.patcher = patch('mymodule.foo')
...         self.mock_foo = self.patcher.start()
...
...     def test_foo(self):
...         self.assertIs(mymodule.foo, self.mock_foo)
...
...     def tearDown(self):
...         self.patcher.stop()
...
>>> MyTest('test_foo').run()
```

If you use this technique you must ensure that the patching is "undone" by calling `stop`. This can be fiddlier than you might think, because if an exception is raised in the setUp then tearDown is not called. *`unittest.TestCase.addCleanup()`* makes this easier :

```
>>> class MyTest(unittest.TestCase):
...     def setUp(self):
...         patcher = patch('mymodule.foo')
...         self.addCleanup(patcher.stop)
...         self.mock_foo = patcher.start()
...
...     def test_foo(self):
...         self.assertIs(mymodule.foo, self.mock_foo)
...
>>> MyTest('test_foo').run()
```

Mocking Unbound Methods

Whilst writing tests today I needed to patch an *unbound method* (patching the method on the class rather than on the instance). I needed self to be passed in as the first argument because I want to make asserts about which objects were calling this particular method. The issue is that you can't patch with a mock for this, because if you replace an unbound method with a mock it doesn't become a bound method when fetched from the instance, and so it doesn't get self passed in. The workaround is to patch the unbound method with a real function instead. The *`patch()`* decorator makes it so simple to patch out methods with a mock that having to create a real function becomes a nuisance.

If you pass `autospec=True` to patch then it does the patching with a *real* function object. This function object has the same signature as the one it is replacing, but delegates to a mock under the hood. You still get your mock auto-created in exactly the same way as before. What it means though, is that if you use it to patch out an unbound method on a class the mocked function will be turned into a bound method if it is fetched from an instance. It will have `self` passed in as the first argument, which is exactly what I wanted :

```
>>> class Foo:
...   def foo(self):
...     pass
...
>>> with patch.object(Foo, 'foo', autospec=True) as mock_foo:
...   mock_foo.return_value = 'foo'
...   foo = Foo()
...   foo.foo()
...
'foo'
>>> mock_foo.assert_called_once_with(foo)
```

If we don't use `autospec=True` then the unbound method is patched out with a Mock instance instead, and isn't called with `self`.

Checking multiple calls with mock

mock has a nice API for making assertions about how your mock objects are used.

```
>>> mock = Mock()
>>> mock.foo_bar.return_value = None
>>> mock.foo_bar('baz', spam='eggs')
>>> mock.foo_bar.assert_called_with('baz', spam='eggs')
```

If your mock is only being called once you can use the *assert_called_once_with()* method that also asserts that the *call_count* is one.

```
>>> mock.foo_bar.assert_called_once_with('baz', spam='eggs')
>>> mock.foo_bar()
>>> mock.foo_bar.assert_called_once_with('baz', spam='eggs')
Traceback (most recent call last):
    ...
AssertionError: Expected 'foo_bar' to be called once. Called 2 times.
Calls: [call('baz', spam='eggs'), call()].
```

Both `assert_called_with` and `assert_called_once_with` make assertions about the *most recent* call. If your mock is going to be called several times, and you want to make assertions about *all* those calls you can use *call_args_list* :

```
>>> mock = Mock(return_value=None)
>>> mock(1, 2, 3)
>>> mock(4, 5, 6)
>>> mock()
>>> mock.call_args_list
[call(1, 2, 3), call(4, 5, 6), call()]
```

The *call* helper makes it easy to make assertions about these calls. You can build up a list of expected calls and compare it to `call_args_list`. This looks remarkably similar to the repr of the `call_args_list` :

```
>>> expected = [call(1, 2, 3), call(4, 5, 6), call()]
>>> mock.call_args_list == expected
True
```

Coping with mutable arguments

Another situation is rare, but can bite you, is when your mock is called with mutable arguments. `call_args` and `call_args_list` store *references* to the arguments. If the arguments are mutated by the code under test then you can no longer make assertions about what the values were when the mock was called.

Here's some example code that shows the problem. Imagine the following functions defined in 'mymodule' :

```
def frob(val):
    pass

def grob(val):
    "First frob and then clear val"
    frob(val)
    val.clear()
```

When we try to test that `grob` calls `frob` with the correct argument look what happens :

```
>>> with patch('mymodule.frob') as mock_frob:
...     val = {6}
...     mymodule.grob(val)
...
>>> val
set()
>>> mock_frob.assert_called_with({6})
Traceback (most recent call last):
    ...
AssertionError: Expected: (({6},), {})
Called with: ((set(),), {})
```

One possibility would be for mock to copy the arguments you pass in. This could then cause problems if you do assertions that rely on object identity for equality.

Here's one solution that uses the *`side_effect`* functionality. If you provide a `side_effect` function for a mock then `side_effect` will be called with the same args as the mock. This gives us an opportunity to copy the arguments and store them for later assertions. In this example I'm using *another* mock to store the arguments so that I can use the mock methods for doing the assertion. Again a helper function sets this up for me.

```
>>> from copy import deepcopy
>>> from unittest.mock import Mock, patch, DEFAULT
>>> def copy_call_args(mock):
...     new_mock = Mock()
...     def side_effect(*args, **kwargs):
...         args = deepcopy(args)
...         kwargs = deepcopy(kwargs)
...         new_mock(*args, **kwargs)
...         return DEFAULT
...     mock.side_effect = side_effect
...     return new_mock
...
>>> with patch('mymodule.frob') as mock_frob:
...     new_mock = copy_call_args(mock_frob)
...     val = {6}
...     mymodule.grob(val)
...
>>> new_mock.assert_called_with({6})
>>> new_mock.call_args
call({6})
```

`copy_call_args` is called with the mock that will be called. It returns a new mock that we do the assertion on. The `side_effect` function makes a copy of the args and calls our `new_mock` with the copy.

Note

If your mock is only going to be used once there is an easier way of checking arguments at the point they are called. You can simply do the checking inside a `side_effect` function.

```
>>> def side_effect(arg):
...     assert arg == {6}
...
>>> mock = Mock(side_effect=side_effect)
>>> mock({6})
>>> mock(set())
Traceback (most recent call last):
    ...
AssertionError
```

An alternative approach is to create a subclass of *Mock* or *MagicMock* that copies (using *copy.deepcopy()*) the arguments. Here's an example implementation :

```
>>> from copy import deepcopy
>>> class CopyingMock(MagicMock):
...     def __call__(self, /, *args, **kwargs):
...         args = deepcopy(args)
...         kwargs = deepcopy(kwargs)
...         return super().__call__(*args, **kwargs)
...
>>> c = CopyingMock(return_value=None)
>>> arg = set()
>>> c(arg)
>>> arg.add(1)
>>> c.assert_called_with(set())
>>> c.assert_called_with(arg)
Traceback (most recent call last):
    ...
AssertionError: expected call not found.
Expected: mock({1})
Actual: mock(set())
>>> c.foo
<CopyingMock name='mock.foo' id='...'>
```

When you subclass `Mock` or `MagicMock` all dynamically created attributes, and the `return_value` will use your subclass automatically. That means all children of a `CopyingMock` will also have the type `CopyingMock`.

Nesting Patches

Using patch as a context manager is nice, but if you do multiple patches you can end up with nested with statements indenting further and further to the right :

```
>>> class MyTest(unittest.TestCase):
...
...     def test_foo(self):
...         with patch('mymodule.Foo') as mock_foo:
...             with patch('mymodule.Bar') as mock_bar:
...                 with patch('mymodule.Spam') as mock_spam:
...                     assert mymodule.Foo is mock_foo
...                     assert mymodule.Bar is mock_bar
...                     assert mymodule.Spam is mock_spam
...
>>> original = mymodule.Foo
>>> MyTest('test_foo').test_foo()
>>> assert mymodule.Foo is original
```

With unittest `cleanup` functions and the *patch methods : start and stop* we can achieve the same effect without the nested indentation. A simple helper method, `create_patch`, puts the patch in place and returns the created mock for us :

```
>>> class MyTest(unittest.TestCase):
...
...     def create_patch(self, name):
...         patcher = patch(name)
...         thing = patcher.start()
...         self.addCleanup(patcher.stop)
...         return thing
...
...     def test_foo(self):
...         mock_foo = self.create_patch('mymodule.Foo')
...         mock_bar = self.create_patch('mymodule.Bar')
...         mock_spam = self.create_patch('mymodule.Spam')
...
...         assert mymodule.Foo is mock_foo
...         assert mymodule.Bar is mock_bar
...         assert mymodule.Spam is mock_spam
...
>>> original = mymodule.Foo
>>> MyTest('test_foo').run()
>>> assert mymodule.Foo is original
```

Mocking a dictionary with MagicMock

You may want to mock a dictionary, or other container object, recording all access to it whilst having it still behave like a dictionary.

We can do this with *`MagicMock`*, which will behave like a dictionary, and using *`side_effect`* to delegate dictionary access to a real underlying dictionary that is under our control.

When the `__getitem__()` and `__setitem__()` methods of our `MagicMock` are called (normal dictionary access) then `side_effect` is called with the key (and in the case of `__setitem__` the value too). We can also control what is returned.

After the `MagicMock` has been used we can use attributes like *`call_args_list`* to assert about how the dictionary was used :

```
>>> my_dict = {'a': 1, 'b': 2, 'c': 3}
>>> def getitem(name):
...      return my_dict[name]
...
>>> def setitem(name, val):
...     my_dict[name] = val
...
>>> mock = MagicMock()
>>> mock.__getitem__.side_effect = getitem
>>> mock.__setitem__.side_effect = setitem
```

Note

An alternative to using `MagicMock` is to use `Mock` and *only* provide the magic methods you specifically want :

```
>>> mock = Mock()
>>> mock.__getitem__ = Mock(side_effect=getitem)
>>> mock.__setitem__ = Mock(side_effect=setitem)
```

A *third* option is to use `MagicMock` but passing in `dict` as the *spec* (or *spec_set*) argument so that the `MagicMock` created only has dictionary magic methods available :

```
>>> mock = MagicMock(spec_set=dict)
>>> mock.__getitem__.side_effect = getitem
>>> mock.__setitem__.side_effect = setitem
```

With these side effect functions in place, the `mock` will behave like a normal dictionary but recording the access. It even raises a `KeyError` if you try to access a key that doesn't exist.

```
>>> mock['a']
1
>>> mock['c']
3
>>> mock['d']
Traceback (most recent call last):
    ...
KeyError: 'd'
>>> mock['b'] = 'fish'
>>> mock['d'] = 'eggs'
>>> mock['b']
'fish'
>>> mock['d']
'eggs'
```

After it has been used you can make assertions about the access using the normal mock methods and attributes :

```
>>> mock.__getitem__.call_args_list
[call('a'), call('c'), call('d'), call('b'), call('d')]
>>> mock.__setitem__.call_args_list
[call('b', 'fish'), call('d', 'eggs')]
>>> my_dict
{'a': 1, 'b': 'fish', 'c': 3, 'd': 'eggs'}
```

Mock subclasses and their attributes

There are various reasons why you might want to subclass `Mock`. One reason might be to add helper methods. Here's a silly example :

```
>>> class MyMock(MagicMock):
...     def has_been_called(self):
...         return self.called
...
>>> mymock = MyMock(return_value=None)
>>> mymock
<MyMock id='...'>
>>> mymock.has_been_called()
False
>>> mymock()
>>> mymock.has_been_called()
True
```

The standard behaviour for `Mock` instances is that attributes and the return value mocks are of the same type as the mock they are accessed on. This ensures that `Mock` attributes are `Mocks` and `MagicMock` attributes are `MagicMocks`[2]. So if you're subclassing to add helper methods then they'll also be available on the attributes and

2. An exception to this rule are the non-callable mocks. Attributes use the callable variant because otherwise non-callable mocks couldn't have callable methods.

return value mock of instances of your subclass.

```
>>> mymock.foo
<MyMock name='mock.foo' id='...'>
>>> mymock.foo.has_been_called()
False
>>> mymock.foo()
<MyMock name='mock.foo()' id='...'>
>>> mymock.foo.has_been_called()
True
```

Sometimes this is inconvenient. For example, one user is subclassing mock to created a Twisted adaptor. Having this applied to attributes too actually causes errors.

`Mock` (in all its flavours) uses a method called `_get_child_mock` to create these "sub-mocks" for attributes and return values. You can prevent your subclass being used for attributes by overriding this method. The signature is that it takes arbitrary keyword arguments (`**kwargs`) which are then passed onto the mock constructor :

```
>>> class Subclass(MagicMock):
...     def _get_child_mock(self, /, **kwargs):
...         return MagicMock(**kwargs)
...
>>> mymock = Subclass()
>>> mymock.foo
<MagicMock name='mock.foo' id='...'>
>>> assert isinstance(mymock, Subclass)
>>> assert not isinstance(mymock.foo, Subclass)
>>> assert not isinstance(mymock(), Subclass)
```

Mocking imports with patch.dict

One situation where mocking can be hard is where you have a local import inside a function. These are harder to mock because they aren't using an object from the module namespace that we can patch out.

Generally local imports are to be avoided. They are sometimes done to prevent circular dependencies, for which there is *usually* a much better way to solve the problem (refactor the code) or to prevent "up front costs" by delaying the import. This can also be solved in better ways than an unconditional local import (store the module as a class or module attribute and only do the import on first use).

That aside there is a way to use `mock` to affect the results of an import. Importing fetches an *object* from the `sys.modules` dictionary. Note that it fetches an *object*, which need not be a module. Importing a module for the first time results in a module object being put in `sys.modules`, so usually when you import something you get a module back. This need not be the case however.

This means you can use `patch.dict()` to *temporarily* put a mock in place in `sys.modules`. Any imports whilst this patch is active will fetch the mock. When the patch is complete (the decorated function exits, the with statement body is complete or `patcher.stop()` is called) then whatever was there previously will be restored safely.

Here's an example that mocks out the 'fooble' module.

```
>>> import sys
>>> mock = Mock()
>>> with patch.dict('sys.modules', {'fooble': mock}):
...     import fooble
...     fooble.blob()
...
<Mock name='mock.blob()' id='...'>
>>> assert 'fooble' not in sys.modules
>>> mock.blob.assert_called_once_with()
```

As you can see the `import fooble` succeeds, but on exit there is no 'fooble' left in `sys.modules`.

This also works for the `from module import name` form :

```
>>> mock = Mock()
>>> with patch.dict('sys.modules', {'fooble': mock}):
...     from fooble import blob
...     blob.blip()
...
<Mock name='mock.blob.blip()' id='...'>
>>> mock.blob.blip.assert_called_once_with()
```

With slightly more work you can also mock package imports :

```
>>> mock = Mock()
>>> modules = {'package': mock, 'package.module': mock.module}
>>> with patch.dict('sys.modules', modules):
...     from package.module import fooble
...     fooble()
...
<Mock name='mock.module.fooble()' id='...'>
>>> mock.module.fooble.assert_called_once_with()
```

Tracking order of calls and less verbose call assertions

The *Mock* class allows you to track the *order* of method calls on your mock objects through the *method_calls* attribute. This doesn't allow you to track the order of calls between separate mock objects, however we can use *mock_calls* to achieve the same effect.

Because mocks track calls to child mocks in `mock_calls`, and accessing an arbitrary attribute of a mock creates a child mock, we can create our separate mocks from a parent one. Calls to those child mock will then all be recorded, in order, in the `mock_calls` of the parent :

```
>>> manager = Mock()
>>> mock_foo = manager.foo
>>> mock_bar = manager.bar
```

```
>>> mock_foo.something()
<Mock name='mock.foo.something()' id='...'>
>>> mock_bar.other.thing()
<Mock name='mock.bar.other.thing()' id='...'>
```

```
>>> manager.mock_calls
[call.foo.something(), call.bar.other.thing()]
```

We can then assert about the calls, including the order, by comparing with the `mock_calls` attribute on the manager mock :

```
>>> expected_calls = [call.foo.something(), call.bar.other.thing()]
>>> manager.mock_calls == expected_calls
True
```

If `patch` is creating, and putting in place, your mocks then you can attach them to a manager mock using the *attach_mock()* method. After attaching calls will be recorded in `mock_calls` of the manager.

```
>>> manager = MagicMock()
>>> with patch('mymodule.Class1') as MockClass1:
...     with patch('mymodule.Class2') as MockClass2:
...         manager.attach_mock(MockClass1, 'MockClass1')
...         manager.attach_mock(MockClass2, 'MockClass2')
```

(suite sur la page suivante)

(suite de la page précédente)

```
...         MockClass1().foo()
...         MockClass2().bar()
<MagicMock name='mock.MockClass1().foo()' id='...'>
<MagicMock name='mock.MockClass2().bar()' id='...'>
>>> manager.mock_calls
[call.MockClass1(),
call.MockClass1().foo(),
call.MockClass2(),
call.MockClass2().bar()]
```

If many calls have been made, but you're only interested in a particular sequence of them then an alternative is to use the *assert_has_calls()* method. This takes a list of calls (constructed with the *call* object). If that sequence of calls are in *mock_calls* then the assert succeeds.

```
>>> m = MagicMock()
>>> m().foo().bar().baz()
<MagicMock name='mock().foo().bar().baz()' id='...'>
>>> m.one().two().three()
<MagicMock name='mock.one().two().three()' id='...'>
>>> calls = call.one().two().three().call_list()
>>> m.assert_has_calls(calls)
```

Even though the chained call `m.one().two().three()` aren't the only calls that have been made to the mock, the assert still succeeds.

Sometimes a mock may have several calls made to it, and you are only interested in asserting about *some* of those calls. You may not even care about the order. In this case you can pass `any_order=True` to `assert_has_calls` :

```
>>> m = MagicMock()
>>> m(1), m.two(2, 3), m.seven(7), m.fifty('50')
(...)
>>> calls = [call.fifty('50'), call(1), call.seven(7)]
>>> m.assert_has_calls(calls, any_order=True)
```

More complex argument matching

Using the same basic concept as *ANY* we can implement matchers to do more complex assertions on objects used as arguments to mocks.

Suppose we expect some object to be passed to a mock that by default compares equal based on object identity (which is the Python default for user defined classes). To use *assert_called_with()* we would need to pass in the exact same object. If we are only interested in some of the attributes of this object then we can create a matcher that will check these attributes for us.

You can see in this example how a 'standard' call to `assert_called_with` isn't sufficient :

```
>>> class Foo:
...     def __init__(self, a, b):
...         self.a, self.b = a, b
...
>>> mock = Mock(return_value=None)
>>> mock(Foo(1, 2))
>>> mock.assert_called_with(Foo(1, 2))
Traceback (most recent call last):
    ...
AssertionError: expected call not found.
Expected: mock(<__main__.Foo object at 0x...>)
Actual: mock(<__main__.Foo object at 0x...>)
```

A comparison function for our `Foo` class might look something like this :

```
>>> def compare(self, other):
...     if not type(self) == type(other):
...         return False
...     if self.a != other.a:
...         return False
...     if self.b != other.b:
...         return False
...     return True
...
```

And a matcher object that can use comparison functions like this for its equality operation would look something like this :

```
>>> class Matcher:
...     def __init__(self, compare, some_obj):
...         self.compare = compare
...         self.some_obj = some_obj
...     def __eq__(self, other):
...         return self.compare(self.some_obj, other)
...
```

Putting all this together :

```
>>> match_foo = Matcher(compare, Foo(1, 2))
>>> mock.assert_called_with(match_foo)
```

The `Matcher` is instantiated with our compare function and the `Foo` object we want to compare against. In `assert_called_with` the `Matcher` equality method will be called, which compares the object the mock was called with against the one we created our matcher with. If they match then `assert_called_with` passes, and if they don't an *`AssertionError`* is raised :

```
>>> match_wrong = Matcher(compare, Foo(3, 4))
>>> mock.assert_called_with(match_wrong)
Traceback (most recent call last):
    ...
AssertionError: Expected: ((<Matcher object at 0x...>,), {})
Called with: ((<Foo object at 0x...>,), {})
```

With a bit of tweaking you could have the comparison function raise the *`AssertionError`* directly and provide a more useful failure message.

As of version 1.5, the Python testing library PyHamcrest provides similar functionality, that may be useful here, in the form of its equality matcher (hamcrest.library.integration.match_equality).

27.8 `test` --- Regression tests package for Python

> **Note**
>
> The *`test`* package is meant for internal use by Python only. It is documented for the benefit of the core developers of Python. Any use of this package outside of Python's standard library is discouraged as code mentioned here can change or be removed without notice between releases of Python.

The *`test`* package contains all regression tests for Python as well as the modules *`test.support`* and *`test.regrtest`*. *`test.support`* is used to enhance your tests while *`test.regrtest`* drives the testing suite.

Each module in the *test* package whose name starts with `test_` is a testing suite for a specific module or feature. All new tests should be written using the *unittest* or *doctest* module. Some older tests are written using a "traditional" testing style that compares output printed to `sys.stdout` ; this style of test is considered deprecated.

Voir aussi

Module *unittest*
Writing PyUnit regression tests.

Module *doctest*
Tests embedded in documentation strings.

27.8.1 Writing Unit Tests for the `test` package

It is preferred that tests that use the *unittest* module follow a few guidelines. One is to name the test module by starting it with `test_` and end it with the name of the module being tested. The test methods in the test module should start with `test_` and end with a description of what the method is testing. This is needed so that the methods are recognized by the test driver as test methods. Also, no documentation string for the method should be included. A comment (such as `# Tests function returns only True or False`) should be used to provide documentation for test methods. This is done because documentation strings get printed out if they exist and thus what test is being run is not stated.

A basic boilerplate is often used :

```
import unittest
from test import support

class MyTestCase1(unittest.TestCase):

    # Only use setUp() and tearDown() if necessary

    def setUp(self):
        ... code to execute in preparation for tests ...

    def tearDown(self):
        ... code to execute to clean up after tests ...

    def test_feature_one(self):
        # Test feature one.
        ... testing code ...

    def test_feature_two(self):
        # Test feature two.
        ... testing code ...

    ... more test methods ...

class MyTestCase2(unittest.TestCase):
    ... same structure as MyTestCase1 ...

... more test classes ...

if __name__ == '__main__':
    unittest.main()
```

This code pattern allows the testing suite to be run by *test.regrtest*, on its own as a script that supports the *unittest* CLI, or via the `python -m unittest` CLI.

The goal for regression testing is to try to break code. This leads to a few guidelines to be followed :

— The testing suite should exercise all classes, functions, and constants. This includes not just the external API that is to be presented to the outside world but also "private" code.
— Whitebox testing (examining the code being tested when the tests are being written) is preferred. Blackbox testing (testing only the published user interface) is not complete enough to make sure all boundary and edge cases are tested.
— Make sure all possible values are tested including invalid ones. This makes sure that not only all valid values are acceptable but also that improper values are handled correctly.
— Exhaust as many code paths as possible. Test where branching occurs and thus tailor input to make sure as many different paths through the code are taken.
— Add an explicit test for any bugs discovered for the tested code. This will make sure that the error does not crop up again if the code is changed in the future.
— Make sure to clean up after your tests (such as close and remove all temporary files).
— If a test is dependent on a specific condition of the operating system then verify the condition already exists before attempting the test.
— Import as few modules as possible and do it as soon as possible. This minimizes external dependencies of tests and also minimizes possible anomalous behavior from side-effects of importing a module.
— Try to maximize code reuse. On occasion, tests will vary by something as small as what type of input is used. Minimize code duplication by subclassing a basic test class with a class that specifies the input :

```
class TestFuncAcceptsSequencesMixin:

    func = mySuperWhammyFunction

    def test_func(self):
        self.func(self.arg)

class AcceptLists(TestFuncAcceptsSequencesMixin, unittest.TestCase):
    arg = [1, 2, 3]

class AcceptStrings(TestFuncAcceptsSequencesMixin, unittest.TestCase):
    arg = 'abc'

class AcceptTuples(TestFuncAcceptsSequencesMixin, unittest.TestCase):
    arg = (1, 2, 3)
```

When using this pattern, remember that all classes that inherit from *unittest.TestCase* are run as tests. The `TestFuncAcceptsSequencesMixin` class in the example above does not have any data and so can't be run by itself, thus it does not inherit from *unittest.TestCase*.

Voir aussi

Test Driven Development
A book by Kent Beck on writing tests before code.

27.8.2 Running tests using the command-line interface

The *test* package can be run as a script to drive Python's regression test suite, thanks to the `-m` option : **python -m test**. Under the hood, it uses *test.regrtest* ; the call **python -m test.regrtest** used in previous Python versions still works. Running the script by itself automatically starts running all regression tests in the *test* package. It does this by finding all modules in the package whose name starts with `test_`, importing them, and executing the function `test_main()` if present or loading the tests via unittest.TestLoader.loadTestsFromModule if `test_main` does not exist. The names of tests to execute may also be passed to the script. Specifying a single regression test (**python -m test test_spam**) will minimize output and only print whether the test passed or failed.

Running *test* directly allows what resources are available for tests to use to be set. You do this by using the `-u` command-line option. Specifying `all` as the value for the `-u` option enables all possible resources : **python -m test -uall**. If all but one resource is desired (a more common case), a comma-separated list of resources that are not desired may be listed after `all`. The command **python -m test -uall,-audio,-largefile** will run

test with all resources except the `audio` and `largefile` resources. For a list of all resources and more command-line options, run **`python -m test -h`**.

Some other ways to execute the regression tests depend on what platform the tests are being executed on. On Unix, you can run **`make test`** at the top-level directory where Python was built. On Windows, executing **`rt.bat`** from your `PCbuild` directory will run all regression tests.

Ajouté dans la version 3.14 : Output is colorized by default and can be controlled using environment variables.

27.9 `test.support` --- Utilities for the Python test suite

The *test.support* module provides support for Python's regression test suite.

> **Note**
>
> *test.support* is not a public module. It is documented here to help Python developers write tests. The API of this module is subject to change without backwards compatibility concerns between releases.

This module defines the following exceptions :

exception `test.support.`**`TestFailed`**

Exception to be raised when a test fails. This is deprecated in favor of *unittest*-based tests and *unittest.TestCase*'s assertion methods.

exception `test.support.`**`ResourceDenied`**

Subclass of *unittest.SkipTest*. Raised when a resource (such as a network connection) is not available. Raised by the *requires()* function.

The *test.support* module defines the following constants :

`test.support.`**`verbose`**

`True` when verbose output is enabled. Should be checked when more detailed information is desired about a running test. *verbose* is set by *test.regrtest*.

`test.support.`**`is_jython`**

`True` if the running interpreter is Jython.

`test.support.`**`is_android`**

`True` if the system is Android.

`test.support.`**`unix_shell`**

Path for shell if not on Windows ; otherwise `None`.

`test.support.`**`LOOPBACK_TIMEOUT`**

Timeout in seconds for tests using a network server listening on the network local loopback interface like `127.0.0.1`.

The timeout is long enough to prevent test failure : it takes into account that the client and the server can run in different threads or even different processes.

The timeout should be long enough for *connect()*, *recv()* and *send()* methods of *socket.socket*.

Its default value is 5 seconds.

See also *INTERNET_TIMEOUT*.

`test.support.`**`INTERNET_TIMEOUT`**

Timeout in seconds for network requests going to the internet.

The timeout is short enough to prevent a test to wait for too long if the internet request is blocked for whatever reason.

Usually, a timeout using *INTERNET_TIMEOUT* should not mark a test as failed, but skip the test instead : see *transient_internet()*.

Its default value is 1 minute.

See also *LOOPBACK_TIMEOUT*.

test.support.**SHORT_TIMEOUT**

Timeout in seconds to mark a test as failed if the test takes "too long".

The timeout value depends on the regrtest `--timeout` command line option.

If a test using *SHORT_TIMEOUT* starts to fail randomly on slow buildbots, use *LONG_TIMEOUT* instead.

Its default value is 30 seconds.

test.support.**LONG_TIMEOUT**

Timeout in seconds to detect when a test hangs.

It is long enough to reduce the risk of test failure on the slowest Python buildbots. It should not be used to mark a test as failed if the test takes "too long". The timeout value depends on the regrtest `--timeout` command line option.

Its default value is 5 minutes.

See also *LOOPBACK_TIMEOUT*, *INTERNET_TIMEOUT* and *SHORT_TIMEOUT*.

test.support.**PGO**

Set when tests can be skipped when they are not useful for PGO.

test.support.**PIPE_MAX_SIZE**

A constant that is likely larger than the underlying OS pipe buffer size, to make writes blocking.

test.support.**Py_DEBUG**

`True` if Python was built with the `Py_DEBUG` macro defined, that is, if Python was built in debug mode.

Ajouté dans la version 3.12.

test.support.**SOCK_MAX_SIZE**

A constant that is likely larger than the underlying OS socket buffer size, to make writes blocking.

test.support.**TEST_SUPPORT_DIR**

Set to the top level directory that contains *test.support*.

test.support.**TEST_HOME_DIR**

Set to the top level directory for the test package.

test.support.**TEST_DATA_DIR**

Set to the `data` directory within the test package.

test.support.**MAX_Py_ssize_t**

Set to *sys.maxsize* for big memory tests.

test.support.**max_memuse**

Set by *set_memlimit()* as the memory limit for big memory tests. Limited by *MAX_Py_ssize_t*.

test.support.**real_max_memuse**

Set by *set_memlimit()* as the memory limit for big memory tests. Not limited by *MAX_Py_ssize_t*.

test.support.**MISSING_C_DOCSTRINGS**

Set to `True` if Python is built without docstrings (the `WITH_DOC_STRINGS` macro is not defined). See the `configure --without-doc-strings` option.

See also the *HAVE_DOCSTRINGS* variable.

test.support.**HAVE_DOCSTRINGS**

Set to `True` if function docstrings are available. See the `python -OO` option, which strips docstrings of functions implemented in Python.

See also the *MISSING_C_DOCSTRINGS* variable.

test.support.**TEST_HTTP_URL**

Define the URL of a dedicated HTTP server for the network tests.

test.support.**ALWAYS_EQ**

Object that is equal to anything. Used to test mixed type comparison.

test.support.**NEVER_EQ**

Object that is not equal to anything (even to *ALWAYS_EQ*). Used to test mixed type comparison.

test.support.**LARGEST**

Object that is greater than anything (except itself). Used to test mixed type comparison.

test.support.**SMALLEST**

Object that is less than anything (except itself). Used to test mixed type comparison.

The *test.support* module defines the following functions :

test.support.**busy_retry**(*timeout*, *err_msg=None*, */*, ***, *error=True*)

Run the loop body until `break` stops the loop.

After *timeout* seconds, raise an *AssertionError* if *error* is true, or just stop the loop if *error* is false.

Example :

```
for _ in support.busy_retry(support.SHORT_TIMEOUT):
    if check():
        break
```

Example of error=False usage :

```
for _ in support.busy_retry(support.SHORT_TIMEOUT, error=False):
    if check():
        break
else:
    raise RuntimeError('my custom error')
```

test.support.**sleeping_retry**(*timeout*, *err_msg=None*, */*, ***, *init_delay=0.010*, *max_delay=1.0*, *error=True*)

Wait strategy that applies exponential backoff.

Run the loop body until `break` stops the loop. Sleep at each loop iteration, but not at the first iteration. The sleep delay is doubled at each iteration (up to *max_delay* seconds).

See *busy_retry()* documentation for the parameters usage.

Example raising an exception after SHORT_TIMEOUT seconds :

```
for _ in support.sleeping_retry(support.SHORT_TIMEOUT):
    if check():
        break
```

Example of error=False usage :

```
for _ in support.sleeping_retry(support.SHORT_TIMEOUT, error=False):
    if check():
        break
else:
    raise RuntimeError('my custom error')
```

`test.support.is_resource_enabled(resource)`

Return `True` if *resource* is enabled and available. The list of available resources is only set when `test.regrtest` is executing the tests.

`test.support.python_is_optimized()`

Return `True` if Python was not built with `-O0` or `-Og`.

`test.support.with_pymalloc()`

Return `_testcapi.WITH_PYMALLOC`.

`test.support.requires(resource, msg=None)`

Raise `ResourceDenied` if *resource* is not available. *msg* is the argument to `ResourceDenied` if it is raised. Always returns `True` if called by a function whose `__name__` is `'__main__'`. Used when tests are executed by `test.regrtest`.

`test.support.sortdict(dict)`

Return a repr of *dict* with keys sorted.

`test.support.findfile(filename, subdir=None)`

Return the path to the file named *filename*. If no match is found *filename* is returned. This does not equal a failure since it could be the path to the file.

Setting *subdir* indicates a relative path to use to find the file rather than looking directly in the path directories.

`test.support.get_pagesize()`

Get size of a page in bytes.

Ajouté dans la version 3.12.

`test.support.setswitchinterval(interval)`

Set the `sys.setswitchinterval()` to the given *interval*. Defines a minimum interval for Android systems to prevent the system from hanging.

`test.support.check_impl_detail(**guards)`

Use this check to guard CPython's implementation-specific tests or to run them only on the implementations guarded by the arguments. This function returns `True` or `False` depending on the host platform. Example usage :

```
check_impl_detail()               # Only on CPython (default).
check_impl_detail(jython=True)    # Only on Jython.
check_impl_detail(cpython=False)  # Everywhere except CPython.
```

`test.support.set_memlimit(limit)`

Set the values for `max_memuse` and `real_max_memuse` for big memory tests.

`test.support.record_original_stdout(stdout)`

Store the value from *stdout*. It is meant to hold the stdout at the time the regrtest began.

`test.support.get_original_stdout()`

Return the original stdout set by `record_original_stdout()` or `sys.stdout` if it's not set.

`test.support.args_from_interpreter_flags()`

Return a list of command line arguments reproducing the current settings in `sys.flags` and `sys.warnoptions`.

`test.support.optim_args_from_interpreter_flags()`

Return a list of command line arguments reproducing the current optimization settings in `sys.flags`.

`test.support.captured_stdin()`

`test.support.captured_stdout()`

test.support.**captured_stderr**()

A context managers that temporarily replaces the named stream with *io.StringIO* object.

Example use with output streams :

```
with captured_stdout() as stdout, captured_stderr() as stderr:
    print("hello")
    print("error", file=sys.stderr)
assert stdout.getvalue() == "hello\n"
assert stderr.getvalue() == "error\n"
```

Example use with input stream :

```
with captured_stdin() as stdin:
    stdin.write('hello\n')
    stdin.seek(0)
    # call test code that consumes from sys.stdin
    captured = input()
self.assertEqual(captured, "hello")
```

test.support.**disable_faulthandler**()

A context manager that temporary disables *faulthandler*.

test.support.**gc_collect**()

Force as many objects as possible to be collected. This is needed because timely deallocation is not guaranteed by the garbage collector. This means that `__del__` methods may be called later than expected and weakrefs may remain alive for longer than expected.

test.support.**disable_gc**()

A context manager that disables the garbage collector on entry. On exit, the garbage collector is restored to its prior state.

test.support.**swap_attr**(*obj*, *attr*, *new_val*)

Context manager to swap out an attribute with a new object.

Utilisation :

```
with swap_attr(obj, "attr", 5):
    ...
```

This will set `obj.attr` to 5 for the duration of the `with` block, restoring the old value at the end of the block. If `attr` doesn't exist on `obj`, it will be created and then deleted at the end of the block.

The old value (or `None` if it doesn't exist) will be assigned to the target of the "as" clause, if there is one.

test.support.**swap_item**(*obj*, *attr*, *new_val*)

Context manager to swap out an item with a new object.

Utilisation :

```
with swap_item(obj, "item", 5):
    ...
```

This will set `obj["item"]` to 5 for the duration of the `with` block, restoring the old value at the end of the block. If `item` doesn't exist on `obj`, it will be created and then deleted at the end of the block.

The old value (or `None` if it doesn't exist) will be assigned to the target of the "as" clause, if there is one.

test.support.**flush_std_streams**()

Call the `flush()` method on *sys.stdout* and then on *sys.stderr*. It can be used to make sure that the logs order is consistent before writing into stderr.

Ajouté dans la version 3.11.

test.support.**print_warning**(*msg*)

Print a warning into `sys.__stderr__`. Format the message as : `f"Warning -- {msg}"`. If *msg* is made of multiple lines, add `"Warning -- "` prefix to each line.

Ajouté dans la version 3.9.

test.support.**wait_process**(*pid*, *, *exitcode*, *timeout=None*)

Wait until process *pid* completes and check that the process exit code is *exitcode*.

Raise an `AssertionError` if the process exit code is not equal to *exitcode*.

If the process runs longer than *timeout* seconds (`SHORT_TIMEOUT` by default), kill the process and raise an `AssertionError`. The timeout feature is not available on Windows.

Ajouté dans la version 3.9.

test.support.**calcobjsize**(*fmt*)

Return the size of the `PyObject` whose structure members are defined by *fmt*. The returned value includes the size of the Python object header and alignment.

test.support.**calcvobjsize**(*fmt*)

Return the size of the `PyVarObject` whose structure members are defined by *fmt*. The returned value includes the size of the Python object header and alignment.

test.support.**checksizeof**(*test*, *o*, *size*)

For testcase *test*, assert that the `sys.getsizeof` for *o* plus the GC header size equals *size*.

@test.support.**anticipate_failure**(*condition*)

A decorator to conditionally mark tests with `unittest.expectedFailure()`. Any use of this decorator should have an associated comment identifying the relevant tracker issue.

test.support.**system_must_validate_cert**(*f*)

A decorator that skips the decorated test on TLS certification validation failures.

@test.support.**run_with_locale**(*catstr*, **locales*)

A decorator for running a function in a different locale, correctly resetting it after it has finished. *catstr* is the locale category as a string (for example `"LC_ALL"`). The *locales* passed will be tried sequentially, and the first valid locale will be used.

@test.support.**run_with_tz**(*tz*)

A decorator for running a function in a specific timezone, correctly resetting it after it has finished.

@test.support.**requires_freebsd_version**(**min_version*)

Decorator for the minimum version when running test on FreeBSD. If the FreeBSD version is less than the minimum, the test is skipped.

@test.support.**requires_linux_version**(**min_version*)

Decorator for the minimum version when running test on Linux. If the Linux version is less than the minimum, the test is skipped.

@test.support.**requires_mac_version**(**min_version*)

Decorator for the minimum version when running test on macOS. If the macOS version is less than the minimum, the test is skipped.

@test.support.**requires_gil_enabled**

Decorator for skipping tests on the free-threaded build. If the *GIL* is disabled, the test is skipped.

@test.support.**requires_IEEE_754**

Decorator for skipping tests on non-IEEE 754 platforms.

@test.support.**requires_zlib**

Decorator for skipping tests if `zlib` doesn't exist.

`@test.support.requires_gzip`

Decorator for skipping tests if `gzip` doesn't exist.

`@test.support.requires_bz2`

Decorator for skipping tests if `bz2` doesn't exist.

`@test.support.requires_lzma`

Decorator for skipping tests if `lzma` doesn't exist.

`@test.support.requires_resource(resource)`

Decorator for skipping tests if *resource* is not available.

`@test.support.requires_docstrings`

Decorator for only running the test if `HAVE_DOCSTRINGS`.

`@test.support.requires_limited_api`

Decorator for only running the test if Limited C API is available.

`@test.support.cpython_only`

Decorator for tests only applicable to CPython.

`@test.support.impl_detail(msg=None, **guards)`

Decorator for invoking `check_impl_detail()` on *guards*. If that returns `False`, then uses *msg* as the reason for skipping the test.

`@test.support.thread_unsafe(reason=None)`

Decorator for marking tests as thread-unsafe. This test always runs in one thread even when invoked with `--parallel-threads`.

`@test.support.no_tracing`

Decorator to temporarily turn off tracing for the duration of the test.

`@test.support.refcount_test`

Decorator for tests which involve reference counting. The decorator does not run the test if it is not run by CPython. Any trace function is unset for the duration of the test to prevent unexpected refcounts caused by the trace function.

`@test.support.bigmemtest(size, memuse, dry_run=True)`

Decorator for bigmem tests.

size is a requested size for the test (in arbitrary, test-interpreted units.) *memuse* is the number of bytes per unit for the test, or a good estimate of it. For example, a test that needs two byte buffers, of 4 GiB each, could be decorated with `@bigmemtest(size=_4G, memuse=2)`.

The *size* argument is normally passed to the decorated test method as an extra argument. If *dry_run* is `True`, the value passed to the test method may be less than the requested value. If *dry_run* is `False`, it means the test doesn't support dummy runs when `-M` is not specified.

`@test.support.bigaddrspacetest`

Decorator for tests that fill the address space.

`test.support.check_syntax_error(testcase, statement, errtext='', *, lineno=None, offset=None)`

Test for syntax errors in *statement* by attempting to compile *statement*. *testcase* is the `unittest` instance for the test. *errtext* is the regular expression which should match the string representation of the raised `SyntaxError`. If *lineno* is not `None`, compares to the line of the exception. If *offset* is not `None`, compares to the offset of the exception.

`test.support.open_urlresource(url, *args, **kw)`

Open *url*. If open fails, raises `TestFailed`.

`test.support.reap_children()`

Use this at the end of `test_main` whenever sub-processes are started. This will help ensure that no extra children (zombies) stick around to hog resources and create problems when looking for refleaks.

test.support.**get_attribute**(*obj*, *name*)

Get an attribute, raising `unittest.SkipTest` if `AttributeError` is raised.

test.support.**catch_unraisable_exception**()

Context manager catching unraisable exception using `sys.unraisablehook()`.

Storing the exception value (`cm.unraisable.exc_value`) creates a reference cycle. The reference cycle is broken explicitly when the context manager exits.

Storing the object (`cm.unraisable.object`) can resurrect it if it is set to an object which is being finalized. Exiting the context manager clears the stored object.

Utilisation :

```
with support.catch_unraisable_exception() as cm:
    # code creating an "unraisable exception"
    ...

    # check the unraisable exception: use cm.unraisable
    ...

# cm.unraisable attribute no longer exists at this point
# (to break a reference cycle)
```

Ajouté dans la version 3.8.

test.support.**load_package_tests**(*pkg_dir*, *loader*, *standard_tests*, *pattern*)

Generic implementation of the `unittest` `load_tests` protocol for use in test packages. *pkg_dir* is the root directory of the package ; *loader*, *standard_tests*, and *pattern* are the arguments expected by `load_tests`. In simple cases, the test package's `__init__.py` can be the following :

```
import os
from test.support import load_package_tests

def load_tests(*args):
    return load_package_tests(os.path.dirname(__file__), *args)
```

test.support.**detect_api_mismatch**(*ref_api*, *other_api*, **, *ignore=()*)

Returns the set of attributes, functions or methods of *ref_api* not found on *other_api*, except for a defined list of items to be ignored in this check specified in *ignore*.

By default this skips private attributes beginning with '_' but includes all magic methods, i.e. those starting and ending in '__'.

Ajouté dans la version 3.5.

test.support.**patch**(*test_instance*, *object_to_patch*, *attr_name*, *new_value*)

Override *object_to_patch.attr_name* with *new_value*. Also add cleanup procedure to *test_instance* to restore *object_to_patch* for *attr_name*. The *attr_name* should be a valid attribute for *object_to_patch*.

test.support.**run_in_subinterp**(*code*)

Run *code* in subinterpreter. Raise `unittest.SkipTest` if `tracemalloc` is enabled.

test.support.**check_free_after_iterating**(*test*, *iter*, *cls*, *args=()*)

Assert instances of *cls* are deallocated after iterating.

test.support.**missing_compiler_executable**(*cmd_names=[]*)

Check for the existence of the compiler executables whose names are listed in *cmd_names* or all the compiler executables when *cmd_names* is empty and return the first missing executable or `None` when none is found missing.

`test.support.check__all__`(*test_case*, *module*, *name_of_module=None*, *extra=()*, *not_exported=()*)

Assert that the `__all__` variable of *module* contains all public names.

The module's public names (its API) are detected automatically based on whether they match the public name convention and were defined in *module*.

The *name_of_module* argument can specify (as a string or tuple thereof) what module(s) an API could be defined in order to be detected as a public API. One case for this is when *module* imports part of its public API from other modules, possibly a C backend (like `csv` and its `_csv`).

The *extra* argument can be a set of names that wouldn't otherwise be automatically detected as "public", like objects without a proper `__module__` attribute. If provided, it will be added to the automatically detected ones.

The *not_exported* argument can be a set of names that must not be treated as part of the public API even though their names indicate otherwise.

Example use :

```
import bar
import foo
import unittest
from test import support

class MiscTestCase(unittest.TestCase):
    def test__all__(self):
        support.check__all__(self, foo)

class OtherTestCase(unittest.TestCase):
    def test__all__(self):
        extra = {'BAR_CONST', 'FOO_CONST'}
        not_exported = {'baz'}  # Undocumented name.
        # bar imports part of its API from _bar.
        support.check__all__(self, bar, ('bar', '_bar'),
                             extra=extra, not_exported=not_exported)
```

Ajouté dans la version 3.6.

`test.support.skip_if_broken_multiprocessing_synchronize`()

Skip tests if the `multiprocessing.synchronize` module is missing, if there is no available semaphore implementation, or if creating a lock raises an `OSError`.

Ajouté dans la version 3.10.

`test.support.check_disallow_instantiation`(*test_case*, *tp*, **args*, ***kwds*)

Assert that type *tp* cannot be instantiated using *args* and *kwds*.

Ajouté dans la version 3.10.

`test.support.adjust_int_max_str_digits`(*max_digits*)

This function returns a context manager that will change the global `sys.set_int_max_str_digits()` setting for the duration of the context to allow execution of test code that needs a different limit on the number of digits when converting between an integer and string.

Ajouté dans la version 3.11.

The `test.support` module defines the following classes :

class `test.support.SuppressCrashReport`

A context manager used to try to prevent crash dialog popups on tests that are expected to crash a subprocess.

On Windows, it disables Windows Error Reporting dialogs using SetErrorMode.

On UNIX, `resource.setrlimit()` is used to set `resource.RLIMIT_CORE`'s soft limit to 0 to prevent coredump file creation.

On both platforms, the old value is restored by `__exit__()`.

class `test.support.`**SaveSignals**

Class to save and restore signal handlers registered by the Python signal handler.

save(*self*)

Save the signal handlers to a dictionary mapping signal numbers to the current signal handler.

restore(*self*)

Set the signal numbers from the `save()` dictionary to the saved handler.

class `test.support.`**Matcher**

matches(*self*, *d*, ***kwargs*)

Try to match a single dict with the supplied arguments.

match_value(*self*, *k*, *dv*, *v*)

Try to match a single stored value (*dv*) with a supplied value (*v*).

27.10 `test.support.socket_helper` --- Utilities for socket tests

The `test.support.socket_helper` module provides support for socket tests.

Ajouté dans la version 3.9.

`test.support.socket_helper.`**IPV6_ENABLED**

Set to `True` if IPv6 is enabled on this host, `False` otherwise.

`test.support.socket_helper.`**find_unused_port**(*family=socket.AF_INET*, *socktype=socket.SOCK_STREAM*)

Returns an unused port that should be suitable for binding. This is achieved by creating a temporary socket with the same family and type as the `sock` parameter (default is `AF_INET`, `SOCK_STREAM`), and binding it to the specified host address (defaults to `0.0.0.0`) with the port set to 0, eliciting an unused ephemeral port from the OS. The temporary socket is then closed and deleted, and the ephemeral port is returned.

Either this method or `bind_port()` should be used for any tests where a server socket needs to be bound to a particular port for the duration of the test. Which one to use depends on whether the calling code is creating a Python socket, or if an unused port needs to be provided in a constructor or passed to an external program (i.e. the `-accept` argument to openssl's s_server mode). Always prefer `bind_port()` over `find_unused_port()` where possible. Using a hard coded port is discouraged since it can make multiple instances of the test impossible to run simultaneously, which is a problem for buildbots.

`test.support.socket_helper.`**bind_port**(*sock*, *host=HOST*)

Bind the socket to a free port and return the port number. Relies on ephemeral ports in order to ensure we are using an unbound port. This is important as many tests may be running simultaneously, especially in a buildbot environment. This method raises an exception if the `sock.family` is `AF_INET` and `sock.type` is `SOCK_STREAM`, and the socket has `SO_REUSEADDR` or `SO_REUSEPORT` set on it. Tests should never set these socket options for TCP/IP sockets. The only case for setting these options is testing multicasting via multiple UDP sockets.

Additionally, if the `SO_EXCLUSIVEADDRUSE` socket option is available (i.e. on Windows), it will be set on the socket. This will prevent anyone else from binding to our host/port for the duration of the test.

`test.support.socket_helper.`**bind_unix_socket**(*sock*, *addr*)

Bind a Unix socket, raising `unittest.SkipTest` if `PermissionError` is raised.

`@test.support.socket_helper.`**skip_unless_bind_unix_socket**

A decorator for running tests that require a functional `bind()` for Unix sockets.

`test.support.socket_helper.`**transient_internet**(*resource_name*, **, timeout=30.0*, *errnos=()*)

A context manager that raises `ResourceDenied` when various issues with the internet connection manifest themselves as exceptions.

27.11 test.support.script_helper --- Utilities for the Python execution tests

The *test.support.script_helper* module provides support for Python's script execution tests.

test.support.script_helper.**interpreter_requires_environment**()

Return True if sys.executable interpreter requires environment variables in order to be able to run at all.

This is designed to be used with @unittest.skipIf() to annotate tests that need to use an assert_python*() function to launch an isolated mode (-I) or no environment mode (-E) sub-interpreter process.

A normal build & test does not run into this situation but it can happen when trying to run the standard library test suite from an interpreter that doesn't have an obvious home with Python's current home finding logic.

Setting PYTHONHOME is one way to get most of the testsuite to run in that situation. PYTHONPATH or PYTHONUSERSITE are other common environment variables that might impact whether or not the interpreter can start.

test.support.script_helper.**run_python_until_end**(**args*, ***env_vars*)

Set up the environment based on *env_vars* for running the interpreter in a subprocess. The values can include __isolated, __cleanenv, __cwd, and TERM.

Modifié dans la version 3.9 : The function no longer strips whitespaces from *stderr*.

test.support.script_helper.**assert_python_ok**(**args*, ***env_vars*)

Assert that running the interpreter with *args* and optional environment variables *env_vars* succeeds (rc == 0) and return a (return code, stdout, stderr) tuple.

If the *__cleanenv* keyword-only parameter is set, *env_vars* is used as a fresh environment.

Python is started in isolated mode (command line option -I), except if the *__isolated* keyword-only parameter is set to False.

Modifié dans la version 3.9 : The function no longer strips whitespaces from *stderr*.

test.support.script_helper.**assert_python_failure**(**args*, ***env_vars*)

Assert that running the interpreter with *args* and optional environment variables *env_vars* fails (rc != 0) and return a (return code, stdout, stderr) tuple.

See *assert_python_ok()* for more options.

Modifié dans la version 3.9 : The function no longer strips whitespaces from *stderr*.

test.support.script_helper.**spawn_python**(**args*, *stdout=subprocess.PIPE*, *stderr=subprocess.STDOUT*, ***kw*)

Run a Python subprocess with the given arguments.

kw is extra keyword args to pass to *subprocess.Popen()*. Returns a *subprocess.Popen* object.

test.support.script_helper.**kill_python**(*p*)

Run the given *subprocess.Popen* process until completion and return stdout.

test.support.script_helper.**make_script**(*script_dir*, *script_basename*, *source*, *omit_suffix=False*)

Create script containing *source* in path *script_dir* and *script_basename*. If *omit_suffix* is False, append .py to the name. Return the full script path.

test.support.script_helper.**make_zip_script**(*zip_dir*, *zip_basename*, *script_name*, *name_in_zip=None*)

Create zip file at *zip_dir* and *zip_basename* with extension zip which contains the files in *script_name*. *name_in_zip* is the archive name. Return a tuple containing (full path, full path of archive name).

test.support.script_helper.**make_pkg**(*pkg_dir*, *init_source=''*)

Create a directory named *pkg_dir* containing an __init__ file with *init_source* as its contents.

test.support.script_helper.**make_zip_pkg**(*zip_dir*, *zip_basename*, *pkg_name*, *script_basename*, *source*, *depth=1*, *compiled=False*)

Create a zip package directory with a path of *zip_dir* and *zip_basename* containing an empty __init__ file and a file *script_basename* containing the *source*. If *compiled* is True, both source files will be compiled and added to the zip package. Return a tuple of the full zip path and the archive name for the zip file.

27.12 test.support.bytecode_helper --- Support tools for testing correct bytecode generation

The *test.support.bytecode_helper* module provides support for testing and inspecting bytecode generation.

Ajouté dans la version 3.9.

The module defines the following class :

class test.support.bytecode_helper.**BytecodeTestCase**(*unittest.TestCase*)

This class has custom assertion methods for inspecting bytecode.

BytecodeTestCase.**get_disassembly_as_string**(*co*)

Return the disassembly of *co* as string.

BytecodeTestCase.**assertInBytecode**(*x*, *opname*, *argval=_UNSPECIFIED*)

Return instr if *opname* is found, otherwise throws *AssertionError*.

BytecodeTestCase.**assertNotInBytecode**(*x*, *opname*, *argval=_UNSPECIFIED*)

Throws *AssertionError* if *opname* is found.

27.13 test.support.threading_helper --- Utilities for threading tests

The *test.support.threading_helper* module provides support for threading tests.

Ajouté dans la version 3.10.

test.support.threading_helper.**join_thread**(*thread*, *timeout=None*)

Join a *thread* within *timeout*. Raise an *AssertionError* if thread is still alive after *timeout* seconds.

@test.support.threading_helper.**reap_threads**

Decorator to ensure the threads are cleaned up even if the test fails.

test.support.threading_helper.**start_threads**(*threads*, *unlock=None*)

Context manager to start *threads*, which is a sequence of threads. *unlock* is a function called after the threads are started, even if an exception was raised ; an example would be *threading.Event.set()*. start_threads will attempt to join the started threads upon exit.

test.support.threading_helper.**threading_cleanup**(**original_values*)

Cleanup up threads not specified in *original_values*. Designed to emit a warning if a test leaves running threads in the background.

test.support.threading_helper.**threading_setup**()

Return current thread count and copy of dangling threads.

test.support.threading_helper.**wait_threads_exit**(*timeout=None*)

Context manager to wait until all threads created in the with statement exit.

test.support.threading_helper.**catch_threading_exception**()

Context manager catching *threading.Thread* exception using *threading.excepthook()*.

Attributes set when an exception is caught :
— exc_type
— exc_value
— exc_traceback
— thread
See *threading.excepthook()* documentation.

These attributes are deleted at the context manager exit.

Utilisation :

```
with threading_helper.catch_threading_exception() as cm:
    # code spawning a thread which raises an exception
    ...

    # check the thread exception, use cm attributes:
    # exc_type, exc_value, exc_traceback, thread
    ...

# exc_type, exc_value, exc_traceback, thread attributes of cm no longer
# exists at this point
# (to avoid reference cycles)
```

Ajouté dans la version 3.8.

27.14 test.support.os_helper --- Utilities for os tests

The *test.support.os_helper* module provides support for os tests.

Ajouté dans la version 3.10.

test.support.os_helper.**FS_NONASCII**

A non-ASCII character encodable by *os.fsencode()*.

test.support.os_helper.**SAVEDCWD**

Set to *os.getcwd()*.

test.support.os_helper.**TESTFN**

Set to a name that is safe to use as the name of a temporary file. Any temporary file that is created should be closed and unlinked (removed).

test.support.os_helper.**TESTFN_NONASCII**

Set to a filename containing the *FS_NONASCII* character, if it exists. This guarantees that if the filename exists, it can be encoded and decoded with the default filesystem encoding. This allows tests that require a non-ASCII filename to be easily skipped on platforms where they can't work.

test.support.os_helper.**TESTFN_UNENCODABLE**

Set to a filename (str type) that should not be able to be encoded by file system encoding in strict mode. It may be `None` if it's not possible to generate such a filename.

test.support.os_helper.**TESTFN_UNDECODABLE**

Set to a filename (bytes type) that should not be able to be decoded by file system encoding in strict mode. It may be `None` if it's not possible to generate such a filename.

test.support.os_helper.**TESTFN_UNICODE**

Set to a non-ASCII name for a temporary file.

class test.support.os_helper.**EnvironmentVarGuard**

Class used to temporarily set or unset environment variables. Instances can be used as a context manager and have a complete dictionary interface for querying/modifying the underlying os.environ. After exit from the context manager all changes to environment variables done through this instance will be rolled back.

Modifié dans la version 3.1 : Added dictionary interface.

class test.support.os_helper.**FakePath**(*path*)

Simple *path-like object*. It implements the *__fspath__()* method which just returns the *path* argument. If *path* is an exception, it will be raised in __fspath__().

EnvironmentVarGuard.**set**(*envvar*, *value*)

Temporarily set the environment variable envvar to the value of value.

EnvironmentVarGuard.**unset**(*envvar*)

Temporarily unset the environment variable envvar.

test.support.os_helper.**can_symlink**()

Return True if the OS supports symbolic links, False otherwise.

test.support.os_helper.**can_xattr**()

Return True if the OS supports xattr, False otherwise.

test.support.os_helper.**change_cwd**(*path*, *quiet=False*)

A context manager that temporarily changes the current working directory to *path* and yields the directory.

If *quiet* is False, the context manager raises an exception on error. Otherwise, it issues only a warning and keeps the current working directory the same.

test.support.os_helper.**create_empty_file**(*filename*)

Create an empty file with *filename*. If it already exists, truncate it.

test.support.os_helper.**fd_count**()

Count the number of open file descriptors.

test.support.os_helper.**fs_is_case_insensitive**(*directory*)

Return True if the file system for *directory* is case-insensitive.

test.support.os_helper.**make_bad_fd**()

Create an invalid file descriptor by opening and closing a temporary file, and returning its descriptor.

test.support.os_helper.**rmdir**(*filename*)

Call *os.rmdir()* on *filename*. On Windows platforms, this is wrapped with a wait loop that checks for the existence of the file, which is needed due to antivirus programs that can hold files open and prevent deletion.

test.support.os_helper.**rmtree**(*path*)

Call *shutil.rmtree()* on *path* or call *os.lstat()* and *os.rmdir()* to remove a path and its contents. As with *rmdir()*, on Windows platforms this is wrapped with a wait loop that checks for the existence of the files.

@test.support.os_helper.**skip_unless_symlink**

A decorator for running tests that require support for symbolic links.

@test.support.os_helper.**skip_unless_xattr**

A decorator for running tests that require support for xattr.

test.support.os_helper.**temp_cwd**(*name='tempcwd'*, *quiet=False*)

A context manager that temporarily creates a new directory and changes the current working directory (CWD).

The context manager creates a temporary directory in the current directory with name *name* before temporarily changing the current working directory. If *name* is None, the temporary directory is created using *tempfile.mkdtemp()*.

If *quiet* is False and it is not possible to create or change the CWD, an error is raised. Otherwise, only a warning is raised and the original CWD is used.

`test.support.os_helper.temp_dir`(*path=None*, *quiet=False*)

A context manager that creates a temporary directory at *path* and yields the directory.

If *path* is `None`, the temporary directory is created using `tempfile.mkdtemp()`. If *quiet* is `False`, the context manager raises an exception on error. Otherwise, if *path* is specified and cannot be created, only a warning is issued.

`test.support.os_helper.temp_umask`(*umask*)

A context manager that temporarily sets the process umask.

`test.support.os_helper.unlink`(*filename*)

Call `os.unlink()` on *filename*. As with `rmdir()`, on Windows platforms, this is wrapped with a wait loop that checks for the existence of the file.

27.15 `test.support.import_helper` --- Utilities for import tests

The `test.support.import_helper` module provides support for import tests.

Ajouté dans la version 3.10.

`test.support.import_helper.forget`(*module_name*)

Remove the module named *module_name* from `sys.modules` and delete any byte-compiled files of the module.

`test.support.import_helper.import_fresh_module`(*name*, *fresh=()*, *blocked=()*, *deprecated=False*)

This function imports and returns a fresh copy of the named Python module by removing the named module from `sys.modules` before doing the import. Note that unlike `reload()`, the original module is not affected by this operation.

fresh is an iterable of additional module names that are also removed from the `sys.modules` cache before doing the import.

blocked is an iterable of module names that are replaced with `None` in the module cache during the import to ensure that attempts to import them raise `ImportError`.

The named module and any modules named in the *fresh* and *blocked* parameters are saved before starting the import and then reinserted into `sys.modules` when the fresh import is complete.

Module and package deprecation messages are suppressed during this import if *deprecated* is `True`.

This function will raise `ImportError` if the named module cannot be imported.

Example use :

```
# Get copies of the warnings module for testing without affecting the
# version being used by the rest of the test suite. One copy uses the
# C implementation, the other is forced to use the pure Python fallback
# implementation
py_warnings = import_fresh_module('warnings', blocked=['_warnings'])
c_warnings = import_fresh_module('warnings', fresh=['_warnings'])
```

Ajouté dans la version 3.1.

`test.support.import_helper.import_module`(*name*, *deprecated=False*, **, required_on=()*)

This function imports and returns the named module. Unlike a normal import, this function raises `unittest.SkipTest` if the module cannot be imported.

Module and package deprecation messages are suppressed during this import if *deprecated* is `True`. If a module is required on a platform but optional for others, set *required_on* to an iterable of platform prefixes which will be compared against `sys.platform`.

Ajouté dans la version 3.1.

`test.support.import_helper.modules_setup()`

Return a copy of `sys.modules`.

`test.support.import_helper.modules_cleanup(oldmodules)`

Remove modules except for *oldmodules* and `encodings` in order to preserve internal cache.

`test.support.import_helper.unload(name)`

Delete *name* from `sys.modules`.

`test.support.import_helper.make_legacy_pyc(source)`

Move a **PEP 3147/PEP 488** pyc file to its legacy pyc location and return the file system path to the legacy pyc file. The *source* value is the file system path to the source file. It does not need to exist, however the PEP 3147/488 pyc file must exist.

class `test.support.import_helper.CleanImport(*module_names)`

A context manager to force import to return a new module reference. This is useful for testing module-level behaviors, such as the emission of a `DeprecationWarning` on import. Example usage :

```
with CleanImport('foo'):
    importlib.import_module('foo')  # New reference.
```

class `test.support.import_helper.DirsOnSysPath(*paths)`

A context manager to temporarily add directories to `sys.path`.

This makes a copy of `sys.path`, appends any directories given as positional arguments, then reverts `sys.path` to the copied settings when the context ends.

Note that *all* `sys.path` modifications in the body of the context manager, including replacement of the object, will be reverted at the end of the block.

27.16 `test.support.warnings_helper` --- Utilities for warnings tests

The `test.support.warnings_helper` module provides support for warnings tests.

Ajouté dans la version 3.10.

`test.support.warnings_helper.ignore_warnings(*, category)`

Suppress warnings that are instances of *category*, which must be `Warning` or a subclass. Roughly equivalent to `warnings.catch_warnings()` with `warnings.simplefilter('ignore', category=category)`. For example :

```
@warning_helper.ignore_warnings(category=DeprecationWarning)
def test_suppress_warning():
    # do something
```

Ajouté dans la version 3.8.

`test.support.warnings_helper.check_no_resource_warning(testcase)`

Context manager to check that no `ResourceWarning` was raised. You must remove the object which may emit `ResourceWarning` before the end of the context manager.

`test.support.warnings_helper.check_syntax_warning(testcase, statement, errtext='', *, lineno=1, offset=None)`

Test for syntax warning in *statement* by attempting to compile *statement*. Test also that the `SyntaxWarning` is emitted only once, and that it will be converted to a `SyntaxError` when turned into error. *testcase* is the `unittest` instance for the test. *errtext* is the regular expression which should match the string representation of the emitted `SyntaxWarning` and raised `SyntaxError`. If *lineno* is not `None`, compares to the line of the warning and exception. If *offset* is not `None`, compares to the offset of the exception.

Ajouté dans la version 3.8.

`test.support.warnings_helper.check_warnings(*filters, quiet=True)`

A convenience wrapper for `warnings.catch_warnings()` that makes it easier to test that a warning was correctly raised. It is approximately equivalent to calling `warnings.catch_warnings(record=True)` with `warnings.simplefilter()` set to `always` and with the option to automatically validate the results that are recorded.

`check_warnings` accepts 2-tuples of the form `("message regexp", WarningCategory)` as positional arguments. If one or more *filters* are provided, or if the optional keyword argument *quiet* is `False`, it checks to make sure the warnings are as expected : each specified filter must match at least one of the warnings raised by the enclosed code or the test fails, and if any warnings are raised that do not match any of the specified filters the test fails. To disable the first of these checks, set *quiet* to `True`.

If no arguments are specified, it defaults to :

```
check_warnings(("", Warning), quiet=True)
```

In this case all warnings are caught and no errors are raised.

On entry to the context manager, a `WarningRecorder` instance is returned. The underlying warnings list from `catch_warnings()` is available via the recorder object's `warnings` attribute. As a convenience, the attributes of the object representing the most recent warning can also be accessed directly through the recorder object (see example below). If no warning has been raised, then any of the attributes that would otherwise be expected on an object representing a warning will return `None`.

The recorder object also has a `reset()` method, which clears the warnings list.

The context manager is designed to be used like this :

```
with check_warnings(("assertion is always true", SyntaxWarning),
                    ("", UserWarning)):
    exec('assert(False, "Hey!")')
    warnings.warn(UserWarning("Hide me!"))
```

In this case if either warning was not raised, or some other warning was raised, `check_warnings()` would raise an error.

When a test needs to look more deeply into the warnings, rather than just checking whether or not they occurred, code like this can be used :

```
with check_warnings(quiet=True) as w:
    warnings.warn("foo")
    assert str(w.args[0]) == "foo"
    warnings.warn("bar")
    assert str(w.args[0]) == "bar"
    assert str(w.warnings[0].args[0]) == "foo"
    assert str(w.warnings[1].args[0]) == "bar"
    w.reset()
    assert len(w.warnings) == 0
```

Here all warnings will be caught, and the test code tests the captured warnings directly.

Modifié dans la version 3.2 : New optional arguments *filters* and *quiet*.

class `test.support.warnings_helper.WarningsRecorder`

Class used to record warnings for unit tests. See documentation of `check_warnings()` above for more details.

CHAPITRE 28

Débogueur et instrumentation

Ces bibliothèques sont là pour vous aider lors du développement en Python : Le débogueur vous permet d'avancer pas à pas dans le code, d'analyser la pile d'appel, de placer des points d'arrêts, … Les outils d'instrumentation exécutent du code et vous donnent un rapport détaillé du temps d'exécution, vous permettant d'identifier les goulots d'étranglement dans vos programmes. Les événements d'audit fournissent une visibilité sur les comportements d'exécution qui nécessiteraient autrement un débogage ou une correction intrusifs.

28.1 Table des évènements d'audit

This table contains all events raised by `sys.audit()` or `PySys_Audit()` calls throughout the CPython runtime and the standard library. These calls were added in 3.8 or later (see **PEP 578**).

Voir `sys.addaudithook()` et `PySys_AddAuditHook()` pour plus d'informations sur la gestion de ces évènements.

Cette table est générée à partir de la documentation de CPython et peut ne pas représenter des évènements levés par d'autres implémentations. Consultez la documentation propre à votre implémentation pour connaître les évènements réellement levés.

Audit event	Arguments
_thread.start_new_thread	`function`, `args`, `kwargs`
array.__new__	`typecode`, `initializer`
builtins.breakpoint	`breakpointhook`
builtins.id	`id`
builtins.input	`prompt`
builtins.input/result	`result`
code.__new__	`code`, `filename`, `name`, `argcount`, `posonlyargcount`, `kwonlyargcount`, `nlocals`, `st`
compile	`source`, `filename`
cpython.PyInterpreterState_Clear	
cpython.PyInterpreterState_New	
cpython._PySys_ClearAuditHooks	
cpython.run_command	`command`
cpython.run_file	`filename`
cpython.run_interactivehook	`hook`
cpython.run_module	`module-name`
cpython.run_startup	`filename`

Tableau 1 – suite de la page précédente

Audit event	Arguments
cpython.run_stdin	
ctypes.addressof	obj
ctypes.call_function	func_pointer, arguments
ctypes.cdata	address
ctypes.cdata/buffer	pointer, size, offset
ctypes.create_string_buffer	init, size
ctypes.create_unicode_buffer	init, size
ctypes.dlopen	name
ctypes.dlsym	library, name
ctypes.dlsym/handle	handle, name
ctypes.get_errno	
ctypes.get_last_error	
ctypes.memoryview_at	address, size, readonly
ctypes.set_errno	errno
ctypes.set_exception	code
ctypes.set_last_error	error
ctypes.string_at	ptr, size
ctypes.wstring_at	ptr, size
ensurepip.bootstrap	root
exec	code_object
fcntl.fcntl	fd, cmd, arg
fcntl.flock	fd, operation
fcntl.ioctl	fd, request, arg
fcntl.lockf	fd, cmd, len, start, whence
ftplib.connect	self, host, port
ftplib.sendcmd	self, cmd
function.__new__	code
gc.get_objects	generation
gc.get_referents	objs
gc.get_referrers	objs
glob.glob	pathname, recursive
glob.glob/2	pathname, recursive, root_dir, dir_fd
http.client.connect	self, host, port
http.client.send	self, data
imaplib.open	self, host, port
imaplib.send	self, data
import	module, filename, sys.path, sys.meta_path, sys.path_hooks
marshal.dumps	value, version
marshal.load	
marshal.loads	bytes
mmap.__new__	fileno, length, access, offset
msvcrt.get_osfhandle	fd
msvcrt.locking	fd, mode, nbytes
msvcrt.open_osfhandle	handle, flags
object.__delattr__	obj, name
object.__getattr__	obj, name
object.__setattr__	obj, name, value
open	path, mode, flags
os.add_dll_directory	path
os.chdir	path
os.chflags	path, flags
os.chmod	path, mode, dir_fd
os.chown	path, uid, gid, dir_fd
os.exec	path, args, env
os.fork	

Tableau 1 – suite de la page précédente

Audit event	Arguments
os.forkpty	
os.fwalk	top, topdown, onerror, follow_symlinks, dir_fd
os.getxattr	path, attribute
os.kill	pid, sig
os.killpg	pgid, sig
os.link	src, dst, src_dir_fd, dst_dir_fd
os.listdir	path
os.listdrives	
os.listmounts	volume
os.listvolumes	
os.listxattr	path
os.lockf	fd, cmd, len
os.mkdir	path, mode, dir_fd
os.posix_spawn	path, argv, env
os.putenv	key, value
os.remove	path, dir_fd
os.removexattr	path, attribute
os.rename	src, dst, src_dir_fd, dst_dir_fd
os.rmdir	path, dir_fd
os.scandir	path
os.setxattr	path, attribute, value, flags
os.spawn	mode, path, args, env
os.startfile	path, operation
os.startfile/2	path, operation, arguments, cwd, show_cmd
os.symlink	src, dst, dir_fd
os.system	command
os.truncate	fd, length
os.unsetenv	key
os.utime	path, times, ns, dir_fd
os.walk	top, topdown, onerror, followlinks
pathlib.Path.glob	self, pattern
pathlib.Path.rglob	self, pattern
pdb.Pdb	
pickle.find_class	module, name
poplib.connect	self, host, port
poplib.putline	self, line
pty.spawn	argv
resource.prlimit	pid, resource, limits
resource.setrlimit	resource, limits
setopencodehook	
shutil.chown	path, user, group
shutil.copyfile	src, dst
shutil.copymode	src, dst
shutil.copystat	src, dst
shutil.copytree	src, dst
shutil.make_archive	base_name, format, root_dir, base_dir
shutil.move	src, dst
shutil.rmtree	path, dir_fd
shutil.unpack_archive	filename, extract_dir, format
signal.pthread_kill	thread_id, signalnum
smtplib.connect	self, host, port
smtplib.send	self, data
socket.__new__	self, family, type, protocol
socket.bind	self, address
socket.connect	self, address

Tableau 1 – suite de la page précédente

Audit event	Arguments
socket.getaddrinfo	`host`, `port`, `family`, `type`, `protocol`
socket.gethostbyaddr	`ip_address`
socket.gethostbyname	`hostname`
socket.gethostname	
socket.getnameinfo	`sockaddr`
socket.getservbyname	`servicename`, `protocolname`
socket.getservbyport	`port`, `protocolname`
socket.sendmsg	`self`, `address`
socket.sendto	`self`, `address`
socket.sethostname	`name`
sqlite3.connect	`database`
sqlite3.connect/handle	`connection_handle`
sqlite3.enable_load_extension	`connection`, `enabled`
sqlite3.load_extension	`connection`, `path`
subprocess.Popen	`executable`, `args`, `cwd`, `env`
sys._current_exceptions	
sys._current_frames	
sys._getframe	`frame`
sys._getframemodulename	`depth`
sys.addaudithook	
sys.excepthook	`hook`, `type`, `value`, `traceback`
sys.set_asyncgen_hooks_finalizer	
sys.set_asyncgen_hooks_firstiter	
sys.setprofile	
sys.settrace	
sys.unraisablehook	`hook`, `unraisable`
syslog.closelog	
syslog.openlog	`ident`, `logoption`, `facility`
syslog.setlogmask	`maskpri`
syslog.syslog	`priority`, `message`
tempfile.mkdtemp	`fullpath`
tempfile.mkstemp	`fullpath`
time.sleep	`secs`
urllib.Request	`fullurl`, `data`, `headers`, `method`
webbrowser.open	`url`
winreg.ConnectRegistry	`computer_name`, `key`
winreg.CreateKey	`key`, `sub_key`, `access`
winreg.DeleteKey	`key`, `sub_key`, `access`
winreg.DeleteValue	`key`, `value`
winreg.DisableReflectionKey	`key`
winreg.EnableReflectionKey	`key`
winreg.EnumKey	`key`, `index`
winreg.EnumValue	`key`, `index`
winreg.ExpandEnvironmentStrings	`str`
winreg.LoadKey	`key`, `sub_key`, `file_name`
winreg.OpenKey	`key`, `sub_key`, `access`
winreg.OpenKey/result	`key`
winreg.PyHKEY.Detach	`key`
winreg.QueryInfoKey	`key`
winreg.QueryReflectionKey	`key`
winreg.QueryValue	`key`, `sub_key`, `value_name`
winreg.SaveKey	`key`, `file_name`
winreg.SetValue	`key`, `sub_key`, `type`, `value`

Les évènements suivants sont levés en interne et ne correspondent à aucune API publique de CPython :

Audit event	Arguments
_winapi.CreateFile	file_name, desired_access, share_mode, creation_disposition, flags_and_attributes
_winapi.CreateJunction	src_path, dst_path
_winapi.CreateNamedPipe	name, open_mode, pipe_mode
_winapi.CreatePipe	
_winapi.CreateProcess	application_name, command_line, current_directory
_winapi.OpenProcess	process_id, desired_access
_winapi.TerminateProcess	handle, exit_code
ctypes.PyObj_FromPtr	obj

28.2 bdb --- Debugger framework

Code source : Lib/bdb.py

The *bdb* module handles basic debugger functions, like setting breakpoints or managing execution via the debugger.

L'exception suivante est définie :

exception bdb.**BdbQuit**

Exception raised by the *Bdb* class for quitting the debugger.

Le module *bdb* définis deux classes :

class bdb.**Breakpoint**(*self*, *file*, *line*, *temporary=False*, *cond=None*, *funcname=None*)

This class implements temporary breakpoints, ignore counts, disabling and (re-)enabling, and conditionals.

Breakpoints are indexed by number through a list called *bpbynumber* and by (file, line) pairs through *bplist*. The former points to a single instance of class *Breakpoint*. The latter points to a list of such instances since there may be more than one breakpoint per line.

When creating a breakpoint, its associated *file name* should be in canonical form. If a *funcname* is defined, a breakpoint *hit* will be counted when the first line of that function is executed. A *conditional* breakpoint always counts a *hit*.

Breakpoint instances have the following methods :

deleteMe()

Delete the breakpoint from the list associated to a file/line. If it is the last breakpoint in that position, it also deletes the entry for the file/line.

enable()

Active le point d'arrêt.

disable()

Désactive le point d'arrêt.

bpformat()

Return a string with all the information about the breakpoint, nicely formatted :
- Breakpoint number.
- Temporary status (del or keep).
- File/line position.
- Break condition.
- Number of times to ignore.

— Number of times hit.
Ajouté dans la version 3.2.

bpprint(*out=None*)

Print the output of `bpformat()` to the file *out*, or if it is `None`, to standard output.

`Breakpoint` instances have the following attributes :

file

File name of the `Breakpoint`.

line

Line number of the `Breakpoint` within `file`.

temporary

`True` if a `Breakpoint` at (file, line) is temporary.

cond

Condition for evaluating a `Breakpoint` at (file, line).

funcname

Function name that defines whether a `Breakpoint` is hit upon entering the function.

enabled

`True` if `Breakpoint` is enabled.

bpbynumber

Numeric index for a single instance of a `Breakpoint`.

bplist

Dictionary of `Breakpoint` instances indexed by (`file`, `line`) tuples.

ignore

Number of times to ignore a `Breakpoint`.

hits

Count of the number of times a `Breakpoint` has been hit.

class bdb.**Bdb**(*skip=None*)

The `Bdb` class acts as a generic Python debugger base class.

This class takes care of the details of the trace facility ; a derived class should implement user interaction. The standard debugger class (`pdb.Pdb`) is an example.

The *skip* argument, if given, must be an iterable of glob-style module name patterns. The debugger will not step into frames that originate in a module that matches one of these patterns. Whether a frame is considered to originate in a certain module is determined by the `__name__` in the frame globals.

Modifié dans la version 3.1 : Added the *skip* parameter.

The following methods of `Bdb` normally don't need to be overridden.

canonic(*filename*)

Return canonical form of *filename*.

For real file names, the canonical form is an operating-system-dependent, `case-normalized absolute path`. A *filename* with angle brackets, such as `"<stdin>"` generated in interactive mode, is returned unchanged.

reset()

Set the `botframe`, `stopframe`, `returnframe` and `quitting` attributes with values ready to start debugging.

trace_dispatch(*frame*, *event*, *arg*)

This function is installed as the trace function of debugged frames. Its return value is the new trace function (in most cases, that is, itself).

The default implementation decides how to dispatch a frame, depending on the type of event (passed as a string) that is about to be executed. *event* can be one of the following :
— `"line"` : A new line of code is going to be executed.
— `"call"` : A function is about to be called, or another code block entered.
— `"return"` : A function or other code block is about to return.
— `"exception"` : Une exception est survenue.
— `"c_call"` : Une fonction C est sur le point d'être appelée.
— `"c_return"` : Une fonction C s'est terminée.
— `"c_exception"` : A C function has raised an exception.
For the Python events, specialized functions (see below) are called. For the C events, no action is taken.

Le paramètre *arg* dépend de l'événement précédent.

See the documentation for `sys.settrace()` for more information on the trace function. For more information on code and frame objects, refer to types.

dispatch_line(*frame*)

If the debugger should stop on the current line, invoke the `user_line()` method (which should be overridden in subclasses). Raise a `BdbQuit` exception if the `quitting` flag is set (which can be set from `user_line()`). Return a reference to the `trace_dispatch()` method for further tracing in that scope.

dispatch_call(*frame*, *arg*)

If the debugger should stop on this function call, invoke the `user_call()` method (which should be overridden in subclasses). Raise a `BdbQuit` exception if the `quitting` flag is set (which can be set from `user_call()`). Return a reference to the `trace_dispatch()` method for further tracing in that scope.

dispatch_return(*frame*, *arg*)

If the debugger should stop on this function return, invoke the `user_return()` method (which should be overridden in subclasses). Raise a `BdbQuit` exception if the `quitting` flag is set (which can be set from `user_return()`). Return a reference to the `trace_dispatch()` method for further tracing in that scope.

dispatch_exception(*frame*, *arg*)

If the debugger should stop at this exception, invokes the `user_exception()` method (which should be overridden in subclasses). Raise a `BdbQuit` exception if the `quitting` flag is set (which can be set from `user_exception()`). Return a reference to the `trace_dispatch()` method for further tracing in that scope.

Normally derived classes don't override the following methods, but they may if they want to redefine the definition of stopping and breakpoints.

is_skipped_line(*module_name*)

Return `True` if *module_name* matches any skip pattern.

stop_here(*frame*)

Return `True` if *frame* is below the starting frame in the stack.

break_here(*frame*)

Return `True` if there is an effective breakpoint for this line.

Check whether a line or function breakpoint exists and is in effect. Delete temporary breakpoints based on information from `effective()`.

break_anywhere(*frame*)

Return `True` if any breakpoint exists for *frame*'s filename.

Derived classes should override these methods to gain control over debugger operation.

user_call(*frame*, *argument_list*)

Called from `dispatch_call()` if a break might stop inside the called function.

argument_list is not used anymore and will always be `None`. The argument is kept for backwards com-

patibility.

user_line(*frame*)

Called from `dispatch_line()` when either `stop_here()` or `break_here()` returns `True`.

user_return(*frame*, *return_value*)

Called from `dispatch_return()` when `stop_here()` returns `True`.

user_exception(*frame*, *exc_info*)

Called from `dispatch_exception()` when `stop_here()` returns `True`.

do_clear(*arg*)

Handle how a breakpoint must be removed when it is a temporary one.

This method must be implemented by derived classes.

Derived classes and clients can call the following methods to affect the stepping state.

set_step()

Arrête après une ligne de code.

set_next(*frame*)

Stop on the next line in or below the given frame.

set_return(*frame*)

Stop when returning from the given frame.

set_until(*frame*, *lineno=None*)

Stop when the line with the *lineno* greater than the current one is reached or when returning from current frame.

set_trace([*frame*])

Start debugging from *frame*. If *frame* is not specified, debugging starts from caller's frame.

Modifié dans la version 3.13 : `set_trace()` will enter the debugger immediately, rather than on the next line of code to be executed.

set_continue()

Stop only at breakpoints or when finished. If there are no breakpoints, set the system trace function to `None`.

set_quit()

Set the `quitting` attribute to `True`. This raises `BdbQuit` in the next call to one of the `dispatch_*()` methods.

Derived classes and clients can call the following methods to manipulate breakpoints. These methods return a string containing an error message if something went wrong, or `None` if all is well.

set_break(*filename*, *lineno*, *temporary=False*, *cond=None*, *funcname=None*)

Set a new breakpoint. If the *lineno* line doesn't exist for the *filename* passed as argument, return an error message. The *filename* should be in canonical form, as described in the `canonic()` method.

clear_break(*filename*, *lineno*)

Delete the breakpoints in *filename* and *lineno*. If none were set, return an error message.

clear_bpbynumber(*arg*)

Delete the breakpoint which has the index *arg* in the `Breakpoint.bpbynumber`. If *arg* is not numeric or out of range, return an error message.

clear_all_file_breaks(*filename*)

Delete all breakpoints in *filename*. If none were set, return an error message.

clear_all_breaks()

Delete all existing breakpoints. If none were set, return an error message.

get_bpbynumber(*arg*)

Return a breakpoint specified by the given number. If *arg* is a string, it will be converted to a number. If *arg* is a non-numeric string, if the given breakpoint never existed or has been deleted, a `ValueError` is raised.

Ajouté dans la version 3.2.

get_break(*filename*, *lineno*)

Return `True` if there is a breakpoint for *lineno* in *filename*.

get_breaks(*filename*, *lineno*)

Return all breakpoints for *lineno* in *filename*, or an empty list if none are set.

get_file_breaks(*filename*)

Return all breakpoints in *filename*, or an empty list if none are set.

get_all_breaks()

Donne tous les points d'arrêt définis.

Derived classes and clients can call the following methods to get a data structure representing a stack trace.

get_stack(*f*, *t*)

Return a list of (frame, lineno) tuples in a stack trace, and a size.

The most recently called frame is last in the list. The size is the number of frames below the frame where the debugger was invoked.

format_stack_entry(*frame_lineno*, *lprefix=': '*)

Return a string with information about a stack entry, which is a `(frame, lineno)` tuple. The return string contains :

— The canonical filename which contains the frame.
— The function name or `"<lambda>"`.
— Les arguments donnés.
— Le résultat.
— La ligne de code (si elle existe).

The following two methods can be called by clients to use a debugger to debug a *statement*, given as a string.

run(*cmd*, *globals=None*, *locals=None*)

Debug a statement executed via the `exec()` function. *globals* defaults to `__main__.__dict__`, *locals* defaults to *globals*.

runeval(*expr*, *globals=None*, *locals=None*)

Debug an expression executed via the `eval()` function. *globals* and *locals* have the same meaning as in `run()`.

runctx(*cmd*, *globals*, *locals*)

For backwards compatibility. Calls the `run()` method.

runcall(*func*, */*, **args*, ***kwds*)

Debug a single function call, and return its result.

Finally, the module defines the following functions :

bdb.**checkfuncname**(*b*, *frame*)

Return `True` if we should break here, depending on the way the `Breakpoint` *b* was set.

If it was set via line number, it checks if `b.line` is the same as the one in *frame*. If the breakpoint was set via `function name`, we have to check we are in the right *frame* (the right function) and if we are on its first executable line.

bdb.**effective**(*file*, *line*, *frame*)

Return `(active breakpoint, delete temporary flag)` or `(None, None)` as the breakpoint to act upon.

The *active breakpoint* is the first entry in `bplist` for the (`file`, `line`) (which must exist) that is `enabled`, for which `checkfuncname()` is true, and that has neither a false `condition` nor positive `ignore` count. The *flag*, meaning that a temporary breakpoint should be deleted, is `False` only when the `cond` cannot be evaluated (in which case, `ignore` count is ignored).

If no such entry exists, then `(None, None)` is returned.

`bdb.set_trace()`

Start debugging with a `Bdb` instance from caller's frame.

28.3 `faulthandler` --- Dump the Python traceback

Ajouté dans la version 3.3.

This module contains functions to dump Python tracebacks explicitly, on a fault, after a timeout, or on a user signal. Call `faulthandler.enable()` to install fault handlers for the `SIGSEGV`, `SIGFPE`, `SIGABRT`, `SIGBUS`, and `SIGILL` signals. You can also enable them at startup by setting the `PYTHONFAULTHANDLER` environment variable or by using the `-X faulthandler` command line option.

The fault handler is compatible with system fault handlers like Apport or the Windows fault handler. The module uses an alternative stack for signal handlers if the `sigaltstack()` function is available. This allows it to dump the traceback even on a stack overflow.

The fault handler is called on catastrophic cases and therefore can only use signal-safe functions (e.g. it cannot allocate memory on the heap). Because of this limitation traceback dumping is minimal compared to normal Python tracebacks :

- Only ASCII is supported. The `backslashreplace` error handler is used on encoding.
- Each string is limited to 500 characters.
- Only the filename, the function name and the line number are displayed. (no source code)
- It is limited to 100 frames and 100 threads.
- The order is reversed : the most recent call is shown first.

By default, the Python traceback is written to `sys.stderr`. To see tracebacks, applications must be run in the terminal. A log file can alternatively be passed to `faulthandler.enable()`.

The module is implemented in C, so tracebacks can be dumped on a crash or when Python is deadlocked.

The *Python Development Mode* calls `faulthandler.enable()` at Python startup.

Voir aussi

Module `pdb`
Interactive source code debugger for Python programs.

Module `traceback`
Standard interface to extract, format and print stack traces of Python programs.

28.3.1 Dumping the traceback

`faulthandler.dump_traceback(file=sys.stderr, all_threads=True)`

Dump the tracebacks of all threads into *file*. If *all_threads* is `False`, dump only the current thread.

Voir aussi

`traceback.print_tb()`, which can be used to print a traceback object.

Modifié dans la version 3.5 : Added support for passing file descriptor to this function.

28.3.2 Fault handler state

`faulthandler.enable`(*file=sys.stderr*, *all_threads=True*)

Enable the fault handler : install handlers for the `SIGSEGV`, `SIGFPE`, `SIGABRT`, `SIGBUS` and `SIGILL` signals to dump the Python traceback. If *all_threads* is `True`, produce tracebacks for every running thread. Otherwise, dump only the current thread.

The *file* must be kept open until the fault handler is disabled : see *issue with file descriptors*.

Modifié dans la version 3.5 : Added support for passing file descriptor to this function.

Modifié dans la version 3.6 : On Windows, a handler for Windows exception is also installed.

Modifié dans la version 3.10 : The dump now mentions if a garbage collector collection is running if *all_threads* is true.

Modifié dans la version 3.14 : Only the current thread is dumped if the *GIL* is disabled to prevent the risk of data races.

`faulthandler.disable`()

Disable the fault handler : uninstall the signal handlers installed by `enable()`.

`faulthandler.is_enabled`()

Check if the fault handler is enabled.

28.3.3 Dumping the tracebacks after a timeout

`faulthandler.dump_traceback_later`(*timeout*, *repeat=False*, *file=sys.stderr*, *exit=False*)

Dump the tracebacks of all threads, after a timeout of *timeout* seconds, or every *timeout* seconds if *repeat* is `True`. If *exit* is `True`, call `_exit()` with status=1 after dumping the tracebacks. (Note `_exit()` exits the process immediately, which means it doesn't do any cleanup like flushing file buffers.) If the function is called twice, the new call replaces previous parameters and resets the timeout. The timer has a sub-second resolution.

The *file* must be kept open until the traceback is dumped or `cancel_dump_traceback_later()` is called : see *issue with file descriptors*.

This function is implemented using a watchdog thread.

Modifié dans la version 3.5 : Added support for passing file descriptor to this function.

Modifié dans la version 3.7 : This function is now always available.

`faulthandler.cancel_dump_traceback_later`()

Cancel the last call to `dump_traceback_later()`.

28.3.4 Dumping the traceback on a user signal

`faulthandler.register`(*signum*, *file=sys.stderr*, *all_threads=True*, *chain=False*)

Register a user signal : install a handler for the *signum* signal to dump the traceback of all threads, or of the current thread if *all_threads* is `False`, into *file*. Call the previous handler if chain is `True`.

The *file* must be kept open until the signal is unregistered by `unregister()` : see *issue with file descriptors*.

Not available on Windows.

Modifié dans la version 3.5 : Added support for passing file descriptor to this function.

`faulthandler.unregister`(*signum*)

Unregister a user signal : uninstall the handler of the *signum* signal installed by `register()`. Return `True` if the signal was registered, `False` otherwise.

Not available on Windows.

28.3.5 Issue with file descriptors

`enable()`, `dump_traceback_later()` and `register()` keep the file descriptor of their *file* argument. If the file is closed and its file descriptor is reused by a new file, or if `os.dup2()` is used to replace the file descriptor, the traceback will be written into a different file. Call these functions again each time that the file is replaced.

28.3.6 Exemple

Example of a segmentation fault on Linux with and without enabling the fault handler :

```
$ python -c "import ctypes; ctypes.string_at(0)"
Segmentation fault

$ python -q -X faulthandler
>>> import ctypes
>>> ctypes.string_at(0)
Fatal Python error: Segmentation fault

Current thread 0x00007fb899f39700 (most recent call first):
  File "/home/python/cpython/Lib/ctypes/__init__.py", line 486 in string_at
  File "<stdin>", line 1 in <module>
Segmentation fault
```

28.4 pdb — Le débogueur Python

Code source : Lib/pdb.py

Le module `pdb` définit un débogueur de code source interactif pour les programmes Python. Il supporte le paramétrage (conditionnel) de points d'arrêt et l'exécution du code source ligne par ligne, l'inspection des *frames* de la pile, la liste du code source, et l'évaluation arbitraire de code Python dans le contexte de n'importe quelle *frame* de la pile. Il supporte aussi le débogage post-mortem et peut être contrôlé depuis un programme.

Le débogueur est extensible -- Il est en réalité défini comme la classe `Pdb`. C'est actuellement non-documenté mais facilement compréhensible en lisant le code source. L'interface d'extension utilise les modules `bdb` et `cmd`.

Voir aussi

Module `faulthandler`
Used to dump Python tracebacks explicitly, on a fault, after a timeout, or on a user signal.

Module `traceback`
Standard interface to extract, format and print stack traces of Python programs.

The typical usage to break into the debugger is to insert :

```
import pdb; pdb.set_trace()
```

Or :

```
breakpoint()
```

at the location you want to break into the debugger, and then run the program. You can then step through the code following this statement, and continue running without the debugger using the `continue` command.

Modifié dans la version 3.7 : The built-in `breakpoint()`, when called with defaults, can be used instead of `import pdb; pdb.set_trace()`.

```
def double(x):
    breakpoint()
    return x * 2
val = 3
print(f"{val} * 2 is {double(val)}")
```

The debugger's prompt is `(Pdb)`, which is the indicator that you are in debug mode :

```
> ...(2)double()
-> breakpoint()
(Pdb) p x
3
(Pdb) continue
3 * 2 is 6
```

Modifié dans la version 3.3 : La complétion via le module *readline* est disponible pour les commandes et les arguments de commande, par exemple les noms *global* et *local* sont proposés comme arguments de la commande `p`.

You can also invoke *pdb* from the command line to debug other scripts. For example :

```
python -m pdb myscript.py
```

When invoked as a module, pdb will automatically enter post-mortem debugging if the program being debugged exits abnormally. After post-mortem debugging (or after normal exit of the program), pdb will restart the program. Automatic restarting preserves pdb's state (such as breakpoints) and in most cases is more useful than quitting the debugger upon program's exit.

Modifié dans la version 3.2 : Added the `-c` option to execute commands as if given in a `.pdbrc` file ; see *Commande du débogueur*.

Modifié dans la version 3.7 : Added the `-m` option to execute modules similar to the way `python -m` does. As with a script, the debugger will pause execution just before the first line of the module.

Typical usage to execute a statement under control of the debugger is :

```
>>> import pdb
>>> def f(x):
...     print(1 / x)
>>> pdb.run("f(2)")
> <string>(1)<module>()
(Pdb) continue
0.5
>>>
```

L'usage typique pour inspecter un programme planté :

```
>>> import pdb
>>> def f(x):
...     print(1 / x)
...
>>> f(0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in f
ZeroDivisionError: division by zero
>>> pdb.pm()
> <stdin>(2)f()
(Pdb) p x
0
(Pdb)
```

Modifié dans la version 3.13 : The implementation of **PEP 667** means that name assignments made via `pdb` will immediately affect the active scope, even when running inside an *optimized scope*.

Le module définit les fonctions suivantes ; chacune entre dans le débogueur d'une manière légèrement différente :

`pdb.run`(*statement*, *globals=None*, *locals=None*)

Exécute la *déclaration* (donnée sous forme de chaîne de caractères ou d'objet code) sous le contrôle du débogueur. L'invite de débogage apparaît avant l'exécution de tout code ; vous pouvez définir des points d'arrêt et taper `continue`, ou vous pouvez passer à travers l'instruction en utilisant `step` ou `next` (toutes ces commandes sont expliquées ci-dessous). Les arguments *globals* et *locals* optionnels spécifient l'environnement dans lequel le code est exécuté ; par défaut le dictionnaire du module `__main__` est utilisé. (Voir l'explication des fonctions natives `exec()` ou `eval()`.)

`pdb.runeval`(*expression*, *globals=None*, *locals=None*)

Evaluate the *expression* (given as a string or a code object) under debugger control. When `runeval()` returns, it returns the value of the *expression*. Otherwise this function is similar to `run()`.

`pdb.runcall`(*function*, **args*, ***kwds*)

Appelle la *function* (une fonction ou une méthode, pas une chaine de caractères) avec les arguments donnés. Quand `runcall()` revient, il retourne ce que l'appel de fonctionne a renvoyé. L'invite de débogage apparaît dès que la fonction est entrée.

`pdb.set_trace`(*, *header=None*, *commands=None*)

Enter the debugger at the calling stack frame. This is useful to hard-code a breakpoint at a given point in a program, even if the code is not otherwise being debugged (e.g. when an assertion fails). If given, *header* is printed to the console just before debugging begins. The *commands* argument, if given, is a list of commands to execute when the debugger starts.

Modifié dans la version 3.7 : L'argument *keyword-only header*.

Modifié dans la version 3.13 : `set_trace()` will enter the debugger immediately, rather than on the next line of code to be executed.

Ajouté dans la version 3.14 : The *commands* argument.

`pdb.post_mortem`(*t=None*)

Enter post-mortem debugging of the given exception or traceback object. If no value is given, it uses the exception that is currently being handled, or raises `ValueError` if there isn't one.

Modifié dans la version 3.13 : Support for exception objects was added.

`pdb.pm`()

Enter post-mortem debugging of the exception found in `sys.last_exc`.

Les fonctions `run*` et `set_trace()` sont des alias pour instancier la classe `Pdb` et appeler la méthode du même nom. Si vous souhaitez accéder à d'autres fonctionnalités, vous devez le faire vous-même :

class `pdb.Pdb`(*completekey='tab'*, *stdin=None*, *stdout=None*, *skip=None*, *nosigint=False*, *readrc=True*, *mode=None*)

Le classe du débogueur est la classe `Pdb`.

Les arguments *completekey*, *stdin* et *stdout* sont transmis à la classe sous-jacente `cmd.Cmd` ; voir la description ici.

L'argument *skip*, s'il est donné, doit être un itérable des noms de modules de style *glob*. Le débogueur n'entrera pas dans les *frames* qui proviennent d'un module qui correspond à l'un de ces motifs. [1]

By default, Pdb sets a handler for the SIGINT signal (which is sent when the user presses `Ctrl-C` on the console) when you give a `continue` command. This allows you to break into the debugger again by pressing `Ctrl-C`. If you want Pdb not to touch the SIGINT handler, set *nosigint* to true.

L'argument *readrc* vaut *True* par défaut et contrôle si Pdb chargera les fichiers *.pdbrc* depuis le système de fichiers.

1. La question de savoir si une *frame* est considérée comme provenant d'un certain module est déterminée par le `__name__` dans les globales de la *frame*.

The *mode* argument specifies how the debugger was invoked. It impacts the workings of some debugger commands. Valid values are `'inline'` (used by the breakpoint() builtin), `'cli'` (used by the command line invocation) or `None` (for backwards compatible behaviour, as before the *mode* argument was added).

Exemple d'appel pour activer le traçage avec *skip* :

```
import pdb; pdb.Pdb(skip=['django.*']).set_trace()
```

Lève un *évènement d'audit* `pdb.Pdb` sans argument.

Modifié dans la version 3.1 : Added the *skip* parameter.

Modifié dans la version 3.2 : Added the *nosigint* parameter. Previously, a SIGINT handler was never set by Pdb.

Modifié dans la version 3.6 : L'argument *readrc*.

Ajouté dans la version 3.14 : Added the *mode* argument.

run(*statement*, *globals=None*, *locals=None*)

runeval(*expression*, *globals=None*, *locals=None*)

runcall(*function*, **args*, ***kwds*)

set_trace()

Voir la documentation pour les fonctions expliquées ci-dessus.

28.4.1 Commande du débogueur

Les commandes reconnues par le débogueur sont listées. La plupart des commandes peuvent être abrégées à une ou deux lettres comme indiquées ; par exemple. `h(elp)` signifie que soit `h` ou `help` peut être utilisée pour entrer la commande *help* (mais pas `he` or `hel`, ni `H` ou `HELP`). Les arguments des commandes doivent être séparées par des espaces (espaces ou tabulations). Les arguments optionnels sont entourés dans des crochets (`[]`) dans la syntaxe de la commande ; les crochets ne doivent pas être insérés. Les alternatives dans la syntaxe de la commande sont séparés par une barre verticale (`|`).

Entrer une ligne vide répète la dernière commande entrée. Exception : si la dernière commande était la commande `list`, les 11 prochaines lignes sont affichées.

Les commandes que le débogueur ne reconnaît pas sont supposées être des instructions Python et sont exécutées dans le contexte du programme en cours de débogage. Les instructions Python peuvent également être préfixées avec un point d'exclamation (`!`). C'est une façon puissante d'inspecter le programme en cours de débogage ; il est même possible de changer une variable ou d'appeler une fonction. Lorsqu'une exception se produit dans une telle instruction, le nom de l'exception est affiché mais l'état du débogueur n'est pas modifié.

Modifié dans la version 3.13 : Expressions/Statements whose prefix is a pdb command are now correctly identified and executed.

Le débogueur supporte *aliases*. Les alias peuvent avoir des paramètres qui permettent un certain niveau d'adaptabilité au contexte étudié.

Multiple commands may be entered on a single line, separated by `;;`. (A single `;` is not used as it is the separator for multiple commands in a line that is passed to the Python parser.) No intelligence is applied to separating the commands ; the input is split at the first `;;` pair, even if it is in the middle of a quoted string. A workaround for strings with double semicolons is to use implicit string concatenation `';'';'` or `";"";"`.

To set a temporary global variable, use a *convenience variable*. A *convenience variable* is a variable whose name starts with `$`. For example, `$foo = 1` sets a global variable `$foo` which you can use in the debugger session. The *convenience variables* are cleared when the program resumes execution so it's less likely to interfere with your program compared to using normal variables like `foo = 1`.

There are three preset *convenience variables* :

- `$_frame` : the current frame you are debugging
- `$_retval` : the return value if the frame is returning
- `$_exception` : the exception if the frame is raising an exception

Ajouté dans la version 3.12 : Added the *convenience variable* feature.

If a file `.pdbrc` exists in the user's home directory or in the current directory, it is read with `'utf-8'` encoding and executed as if it had been typed at the debugger prompt, with the exception that empty lines and lines starting with `#` are ignored. This is particularly useful for aliases. If both files exist, the one in the home directory is read first and aliases defined there can be overridden by the local file.

Modifié dans la version 3.2 : Le fichier `.pdbrc` peut maintenant contenir des commandes qui continue le débogage, comme `continue` ou `next`. Précédemment, ces commandes n'avaient aucun effet.

Modifié dans la version 3.11 : `.pdbrc` is now read with `'utf-8'` encoding. Previously, it was read with the system locale encoding.

h(elp) `[command]`

Sans argument, affiche la liste des commandes disponibles. Avec une *commande* comme argument, affiche l'aide de cette commande. `help pdb` affiche la documentation complète (la *docstring* du module `pdb`). Puisque l'argument *command* doit être un identificateur, `help exec` doit être entré pour obtenir de l'aide sur la commande `!`.

w(here) `[count]`

Print a stack trace, with the most recent frame at the bottom. if *count* is 0, print the current frame entry. If *count* is negative, print the least recent - *count* frames. If *count* is positive, print the most recent *count* frames. An arrow (>) indicates the current frame, which determines the context of most commands.

Modifié dans la version 3.14 : *count* argument is added.

d(own) `[count]`

Déplace le niveau de la *frame* courante *count* (par défaut un) vers le bas dans la trace de pile (vers une *frame* plus récente).

u(p) `[count]`

Déplace le niveau de la *frame* courante *count* (par défaut un) vers le haut dans la trace de pile (vers une *frame* plus ancienne).

b(reak) `[([filename:]lineno | function) [, condition]]`

With a *lineno* argument, set a break at line *lineno* in the current file. The line number may be prefixed with a *filename* and a colon, to specify a breakpoint in another file (possibly one that hasn't been loaded yet). The file is searched on `sys.path`. Acceptable forms of *filename* are `/abspath/to/file.py`, `relpath/file.py`, `module` and `package.module`.

With a *function* argument, set a break at the first executable statement within that function. *function* can be any expression that evaluates to a function in the current namespace.

Si un second argument est présent, c'est une expression qui doit évaluer à *True* avant que le point d'arrêt ne soit honoré.

Sans argument, liste tous les arrêts, incluant pour chaque point d'arrêt, le nombre de fois qu'un point d'arrêt a été atteint, le nombre de ignore, et la condition associée le cas échéant.

Each breakpoint is assigned a number to which all the other breakpoint commands refer.

tbreak `[([filename:]lineno | function) [, condition]]`

Point d'arrêt temporaire, qui est enlevé automatiquement au premier passage. Les arguments sont les mêmes que pour `break`.

cl(ear) `[filename:lineno | bpnumber ...]`

Avec un argument *filename :lineno*, efface tous les points d'arrêt sur cette ligne. Avec une liste de numéros de points d'arrêt séparés par un espace, efface ces points d'arrêt. Sans argument, efface tous les points d'arrêt (mais demande d'abord confirmation).

disable `bpnumber [bpnumber ...]`

Désactive les points d'arrêt indiqués sous la forme d'une liste de numéros de points d'arrêt séparés par un espace. Désactiver un point d'arrêt signifie qu'il ne peut pas interrompre l'exécution du programme, mais à la différence d'effacer un point d'arrêt, il reste dans la liste des points d'arrêt et peut être (ré)activé.

enable bpnumber [bpnumber ...]

Active les points d'arrêt spécifiés.

ignore bpnumber [count]

Set the ignore count for the given breakpoint number. If *count* is omitted, the ignore count is set to 0. A breakpoint becomes active when the ignore count is zero. When non-zero, the *count* is decremented each time the breakpoint is reached and the breakpoint is not disabled and any associated condition evaluates to true.

condition bpnumber [condition]

Définit une nouvelle *condition* pour le point d'arrêt, une expression qui doit évaluer à *True* avant que le point d'arrêt ne soit honoré. Si *condition* est absente, toute condition existante est supprimée, c'est-à-dire que le point d'arrêt est rendu inconditionnel.

commands [bpnumber]

Spécifie une liste de commandes pour le numéro du point d'arrêt *bpnumber*. Les commandes elles-mêmes apparaissent sur les lignes suivantes. Tapez une ligne contenant juste `end` pour terminer les commandes. Un exemple :

```
(Pdb) commands 1
(com) p some_variable
(com) end
(Pdb)
```

Pour supprimer toutes les commandes depuis un point d'arrêt, écrivez `commands` suivi immédiatement de `end` ; ceci supprime les commandes.

Sans argument *bpnumber*, `commands` se réfère au dernier point d'arrêt défini.

Vous pouvez utiliser les commandes de point d'arrêt pour redémarrer votre programme. Utilisez simplement la commande `continue`, ou `step`, ou toute autre commande qui reprend l'exécution.

Specifying any command resuming execution (currently `continue`, `step`, `next`, `return`, `until`, `jump`, `quit` and their abbreviations) terminates the command list (as if that command was immediately followed by end). This is because any time you resume execution (even with a simple next or step), you may encounter another breakpoint—which could have its own command list, leading to ambiguities about which list to execute.

If the list of commands contains the `silent` command, or a command that resumes execution, then the breakpoint message containing information about the frame is not displayed.

Modifié dans la version 3.14 : Frame information will not be displayed if a command that resumes execution is present in the command list.

s(tep)

Exécute la ligne en cours, s'arrête à la première occasion possible (soit dans une fonction qui est appelée, soit sur la ligne suivante de la fonction courante).

n(ext)

Continue l'exécution jusqu'à ce que la ligne suivante de la fonction en cours soit atteinte ou qu'elle revienne. (La différence entre `next` et `step` est que `step` s'arrête dans une fonction appelée, tandis que `next` exécute les fonctions appelées à (presque) pleine vitesse, ne s'arrêtant qu'à la ligne suivante dans la fonction courante.)

unt(il) [lineno]

Sans argument, continue l'exécution jusqu'à ce que la ligne avec un nombre supérieur au nombre actuel soit atteinte.

With *lineno*, continue execution until a line with a number greater or equal to *lineno* is reached. In both cases, also stop when the current frame returns.

Modifié dans la version 3.2 : Permet de donner un numéro de ligne explicite.

r(eturn)

Continue l'exécution jusqu'au retour de la fonction courante.

c(ont(inue))

Continue l'exécution, seulement s'arrête quand un point d'arrêt est rencontré.

j(ump) `lineno`

Définit la prochaine ligne qui sera exécutée. Uniquement disponible dans la *frame* inférieur. Cela vous permet de revenir en arrière et d'exécuter à nouveau le code, ou de passer en avant pour sauter le code que vous ne voulez pas exécuter.

Il est à noter que tous les sauts ne sont pas autorisés -- par exemple, il n'est pas possible de sauter au milieu d'une boucle `for` ou en dehors d'une clause `finally`.

l(ist) `[first[, last]]`

Liste le code source du fichier courant. Sans arguments, liste 11 lignes autour de la ligne courante ou continue le listing précédant. Avec l'argument `.`, liste 11 lignes autour de la ligne courante. Avec un argument, liste les 11 lignes autour de cette ligne. Avec deux arguments, liste la plage donnée ; si le second argument est inférieur au premier, il est interprété comme un compte.

La ligne en cours dans l'image courante est indiquée par `->`. Si une exception est en cours de débogage, la ligne où l'exception a été initialement levée ou propagée est indiquée par `>>`, si elle diffère de la ligne courante.

Modifié dans la version 3.2 : Added the `>>` marker.

ll | `longlist`

Liste le code source de la fonction ou du bloc courant. Les lignes intéressantes sont marquées comme pour *`list`*.

Ajouté dans la version 3.2.

a(rgs)

Print the arguments of the current function and their current values.

p `expression`

Evaluate *expression* in the current context and print its value.

> **Note**
>
> `print()` peut aussi être utilisée, mais n'est pas une commande du débogueur --- il exécute la fonction Python *`print()`*.

pp `expression`

Like the *`p`* command, except the value of *expression* is pretty-printed using the *`pprint`* module.

whatis `expression`

Print the type of *expression*.

source `expression`

Try to get source code of *expression* and display it.

Ajouté dans la version 3.2.

display `[expression]`

Display the value of *expression* if it changed, each time execution stops in the current frame.

Without *expression*, list all display expressions for the current frame.

> **Note**
>
> Display evaluates *expression* and compares to the result of the previous evaluation of *expression*, so when the result is mutable, display may not be able to pick up the changes.

Exemple :

```
lst = []
breakpoint()
pass
lst.append(1)
print(lst)
```

Display won't realize `lst` has been changed because the result of evaluation is modified in place by `lst.append(1)` before being compared :

```
> example.py(3)<module>()
-> pass
(Pdb) display lst
display lst: []
(Pdb) n
> example.py(4)<module>()
-> lst.append(1)
(Pdb) n
> example.py(5)<module>()
-> print(lst)
(Pdb)
```

You can do some tricks with copy mechanism to make it work :

```
> example.py(3)<module>()
-> pass
(Pdb) display lst[:]
display lst[:]: []
(Pdb) n
> example.py(4)<module>()
-> lst.append(1)
(Pdb) n
> example.py(5)<module>()
-> print(lst)
display lst[:]: [1]  [old: []]
(Pdb)
```

Ajouté dans la version 3.2.

undisplay `[expression]`

Do not display *expression* anymore in the current frame. Without *expression*, clear all display expressions for the current frame.

Ajouté dans la version 3.2.

interact

Start an interactive interpreter (using the *code* module) in a new global namespace initialised from the local and global namespaces for the current scope. Use `exit()` or `quit()` to exit the interpreter and return to the debugger.

> **Note**
>
> As `interact` creates a new dedicated namespace for code execution, assignments to variables will not affect the original namespaces. However, modifications to any referenced mutable objects will be reflected in the original namespaces as usual.

Ajouté dans la version 3.2.

Modifié dans la version 3.13 : `exit()` and `quit()` can be used to exit the *interact* command.

Modifié dans la version 3.13 : `interact` directs its output to the debugger's output channel rather than `sys.stderr`.

alias [name [command]]

Create an alias called *name* that executes *command*. The *command* must *not* be enclosed in quotes. Replaceable parameters can be indicated by `%1`, `%2`, ... and `%9`, while `%*` is replaced by all the parameters. If *command* is omitted, the current alias for *name* is shown. If no arguments are given, all aliases are listed.

Les alias peuvent être imbriqués et peuvent contenir tout ce qui peut être légalement tapé à l'invite *pdb*. Notez que les commandes *pdb* internes *peuvent* être remplacées par des alias. Une telle commande est alors masquée jusqu'à ce que l'alias soit supprimé. L'*aliasing* est appliqué récursivement au premier mot de la ligne de commande ; tous les autres mots de la ligne sont laissés seuls.

Comme un exemple, voici deux alias utiles (spécialement quand il est placé dans le fichier `.pdbrc`) :

```
# Print instance variables (usage "pi classInst")
alias pi for k in %1.__dict__.keys(): print(f"%1.{k} = {%1.__dict__[k]}")
# Print instance variables in self
alias ps pi self
```

unalias name

Delete the specified alias *name*.

! statement

Execute the (one-line) *statement* in the context of the current stack frame. The exclamation point can be omitted unless the first word of the statement resembles a debugger command, e.g. :

```
(Pdb) ! n=42
(Pdb)
```

To set a global variable, you can prefix the assignment command with a `global` statement on the same line, e.g. :

```
(Pdb) global list_options; list_options = ['-l']
(Pdb)
```

run [args ...]

restart [args ...]

Restart the debugged Python program. If *args* is supplied, it is split with `shlex` and the result is used as the new `sys.argv`. History, breakpoints, actions and debugger options are preserved. `restart` is an alias for `run`.

Modifié dans la version 3.14 : `run` and `restart` commands are disabled when the debugger is invoked in `'inline'` mode.

q(uit)

Quit from the debugger. The program being executed is aborted. An end-of-file input is equivalent to `quit`.

A confirmation prompt will be shown if the debugger is invoked in `'inline'` mode. Either `y`, `Y`, `<Enter>` or `EOF` will confirm the quit.

Modifié dans la version 3.14 : A confirmation prompt will be shown if the debugger is invoked in `'inline'` mode. After the confirmation, the debugger will call `sys.exit()` immediately, instead of raising `bdb.BdbQuit` in the next trace event.

debug code

Enter a recursive debugger that steps through *code* (which is an arbitrary expression or statement to be executed in the current environment).

retval

Print the return value for the last return of the current function.

exceptions [excnumber]

List or jump between chained exceptions.

When using `pdb.pm()` or `Pdb.post_mortem(...)` with a chained exception instead of a traceback, it allows the user to move between the chained exceptions using `exceptions` command to list exceptions, and `exception <number>` to switch to that exception.

Exemple :

```
def out():
    try:
        middle()
    except Exception as e:
        raise ValueError("reraise middle() error") from e

def middle():
    try:
        return inner(0)
    except Exception as e:
        raise ValueError("Middle fail")

def inner(x):
    1 / x

 out()
```

calling `pdb.pm()` will allow to move between exceptions :

```
> example.py(5)out()
-> raise ValueError("reraise middle() error") from e

(Pdb) exceptions
  0 ZeroDivisionError('division by zero')
  1 ValueError('Middle fail')
> 2 ValueError('reraise middle() error')

(Pdb) exceptions 0
> example.py(16)inner()
-> 1 / x

(Pdb) up
> example.py(10)middle()
-> return inner(0)
```

Ajouté dans la version 3.13.

Notes

28.5 The Python Profilers

Source code : Lib/profile.py and Lib/pstats.py

28.5.1 Introduction to the profilers

cProfile and *profile* provide *deterministic profiling* of Python programs. A *profile* is a set of statistics that describes how often and for how long various parts of the program executed. These statistics can be formatted into reports via the *pstats* module.

The Python standard library provides two different implementations of the same profiling interface :

1. *cProfile* is recommended for most users ; it's a C extension with reasonable overhead that makes it suitable for profiling long-running programs. Based on `lsprof`, contributed by Brett Rosen and Ted Czotter.
2. *profile*, a pure Python module whose interface is imitated by *cProfile*, but which adds significant overhead to profiled programs. If you're trying to extend the profiler in some way, the task might be easier with this module. Originally designed and written by Jim Roskind.

> **Note**
>
> The profiler modules are designed to provide an execution profile for a given program, not for benchmarking purposes (for that, there is *timeit* for reasonably accurate results). This particularly applies to benchmarking Python code against C code : the profilers introduce overhead for Python code, but not for C-level functions, and so the C code would seem faster than any Python one.

28.5.2 Instant User's Manual

This section is provided for users that "don't want to read the manual." It provides a very brief overview, and allows a user to rapidly perform profiling on an existing application.

To profile a function that takes a single argument, you can do :

```
import cProfile
import re
cProfile.run('re.compile("foo|bar")')
```

(Use *profile* instead of *cProfile* if the latter is not available on your system.)

The above action would run *re.compile()* and print profile results like the following :

```
      214 function calls (207 primitive calls) in 0.002 seconds

Ordered by: cumulative time

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.000    0.000    0.002    0.002 {built-in method builtins.exec}
     1    0.000    0.000    0.001    0.001 <string>:1(<module>)
     1    0.000    0.000    0.001    0.001 __init__.py:250(compile)
     1    0.000    0.000    0.001    0.001 __init__.py:289(_compile)
     1    0.000    0.000    0.000    0.000 _compiler.py:759(compile)
     1    0.000    0.000    0.000    0.000 _parser.py:937(parse)
     1    0.000    0.000    0.000    0.000 _compiler.py:598(_code)
     1    0.000    0.000    0.000    0.000 _parser.py:435(_parse_sub)
```

The first line indicates that 214 calls were monitored. Of those calls, 207 were *primitive*, meaning that the call was not induced via recursion. The next line : `Ordered by: cumulative time` indicates the output is sorted by the `cumtime` values. The column headings include :

ncalls
for the number of calls.

tottime
for the total time spent in the given function (and excluding time made in calls to sub-functions)

percall
is the quotient of `tottime` divided by `ncalls`

cumtime
is the cumulative time spent in this and all subfunctions (from invocation till exit). This figure is accurate *even* for recursive functions.

percall
is the quotient of `cumtime` divided by primitive calls

filename :lineno(function)
provides the respective data of each function

When there are two numbers in the first column (for example `3/1`), it means that the function recursed. The second value is the number of primitive calls and the former is the total number of calls. Note that when the function does not recurse, these two values are the same, and only the single figure is printed.

Instead of printing the output at the end of the profile run, you can save the results to a file by specifying a filename to the `run()` function :

```
import cProfile
import re
cProfile.run('re.compile("foo|bar")', 'restats')
```

The *pstats.Stats* class reads profile results from a file and formats them in various ways.

The files *cProfile* and *profile* can also be invoked as a script to profile another script. For example :

```
python -m cProfile [-o output_file] [-s sort_order] (-m module | myscript.py)
```

`-o` writes the profile results to a file instead of to stdout

`-s` specifies one of the *sort_stats()* sort values to sort the output by. This only applies when `-o` is not supplied.

`-m` specifies that a module is being profiled instead of a script.

Ajouté dans la version 3.7 : Added the `-m` option to *cProfile*.

Ajouté dans la version 3.8 : Added the `-m` option to *profile*.

The *pstats* module's *Stats* class has a variety of methods for manipulating and printing the data saved into a profile results file :

```
import pstats
from pstats import SortKey
p = pstats.Stats('restats')
p.strip_dirs().sort_stats(-1).print_stats()
```

The *strip_dirs()* method removed the extraneous path from all the module names. The *sort_stats()* method sorted all the entries according to the standard module/line/name string that is printed. The *print_stats()* method printed out all the statistics. You might try the following sort calls :

```
p.sort_stats(SortKey.NAME)
p.print_stats()
```

The first call will actually sort the list by function name, and the second call will print out the statistics. The following are some interesting calls to experiment with :

```
p.sort_stats(SortKey.CUMULATIVE).print_stats(10)
```

This sorts the profile by cumulative time in a function, and then only prints the ten most significant lines. If you want to understand what algorithms are taking time, the above line is what you would use.

If you were looking to see what functions were looping a lot, and taking a lot of time, you would do :

```
p.sort_stats(SortKey.TIME).print_stats(10)
```

to sort according to time spent within each function, and then print the statistics for the top ten functions.

You might also try :

```
p.sort_stats(SortKey.FILENAME).print_stats('__init__')
```

This will sort all the statistics by file name, and then print out statistics for only the class init methods (since they are spelled with `__init__` in them). As one final example, you could try :

```
p.sort_stats(SortKey.TIME, SortKey.CUMULATIVE).print_stats(.5, 'init')
```

This line sorts statistics with a primary key of time, and a secondary key of cumulative time, and then prints out some of the statistics. To be specific, the list is first culled down to 50% (re : `.5`) of its original size, then only lines containing `init` are maintained, and that sub-sub-list is printed.

If you wondered what functions called the above functions, you could now (`p` is still sorted according to the last criteria) do :

```
p.print_callers(.5, 'init')
```

and you would get a list of callers for each of the listed functions.

If you want more functionality, you're going to have to read the manual, or guess what the following functions do :

```
p.print_callees()
p.add('restats')
```

Invoked as a script, the *pstats* module is a statistics browser for reading and examining profile dumps. It has a simple line-oriented interface (implemented using *cmd*) and interactive help.

28.5.3 `profile` and `cProfile` Module Reference

Both the *profile* and *cProfile* modules provide the following functions :

`profile.`**`run`**(*command*, *filename=None*, *sort=-1*)

This function takes a single argument that can be passed to the *exec()* function, and an optional file name. In all cases this routine executes :

```
exec(command, __main__.__dict__, __main__.__dict__)
```

and gathers profiling statistics from the execution. If no file name is present, then this function automatically creates a *Stats* instance and prints a simple profiling report. If the sort value is specified, it is passed to this *Stats* instance to control how the results are sorted.

`profile.`**`runctx`**(*command*, *globals*, *locals*, *filename=None*, *sort=-1*)

This function is similar to *run()*, with added arguments to supply the globals and locals mappings for the *command* string. This routine executes :

```
exec(command, globals, locals)
```

and gathers profiling statistics as in the *run()* function above.

`class` `profile.`**`Profile`**(*timer=None*, *timeunit=0.0*, *subcalls=True*, *builtins=True*)

This class is normally only used if more precise control over profiling is needed than what the `cProfile.run()` function provides.

A custom timer can be supplied for measuring how long code takes to run via the *timer* argument. This must be a function that returns a single number representing the current time. If the number is an integer, the *timeunit* specifies a multiplier that specifies the duration of each unit of time. For example, if the timer returns times measured in thousands of seconds, the time unit would be `.001`.

Directly using the *Profile* class allows formatting profile results without writing the profile data to a file :

```
import cProfile, pstats, io
from pstats import SortKey
pr = cProfile.Profile()
pr.enable()
# ... do something ...
pr.disable()
s = io.StringIO()
```

(suite sur la page suivante)

(suite de la page précédente)

```
sortby = SortKey.CUMULATIVE
ps = pstats.Stats(pr, stream=s).sort_stats(sortby)
ps.print_stats()
print(s.getvalue())
```

The `Profile` class can also be used as a context manager (supported only in `cProfile` module. see *Le type gestionnaire de contexte*) :

```
import cProfile

with cProfile.Profile() as pr:
    # ... do something ...

    pr.print_stats()
```

Modifié dans la version 3.8 : Ajout de la gestion des gestionnaires de contexte.

enable()

Start collecting profiling data. Only in `cProfile`.

disable()

Stop collecting profiling data. Only in `cProfile`.

create_stats()

Stop collecting profiling data and record the results internally as the current profile.

print_stats(*sort=-1*)

Create a `Stats` object based on the current profile and print the results to stdout.

The *sort* parameter specifies the sorting order of the displayed statistics. It accepts a single key or a tuple of keys to enable multi-level sorting, as in `Stats.sort_stats`.

Ajouté dans la version 3.13 : `print_stats()` now accepts a tuple of keys.

dump_stats(*filename*)

Write the results of the current profile to *filename*.

run(*cmd*)

Profile the cmd via `exec()`.

runctx(*cmd*, *globals*, *locals*)

Profile the cmd via `exec()` with the specified global and local environment.

runcall(*func*, /, **args*, ***kwargs*)

Profile `func(*args, **kwargs)`

Note that profiling will only work if the called command/function actually returns. If the interpreter is terminated (e.g. via a `sys.exit()` call during the called command/function execution) no profiling results will be printed.

28.5.4 The `Stats` Class

Analysis of the profiler data is done using the `Stats` class.

class pstats.**Stats**(**filenames or profile*, *stream=sys.stdout*)

This class constructor creates an instance of a "statistics object" from a *filename* (or list of filenames) or from a `Profile` instance. Output will be printed to the stream specified by *stream*.

The file selected by the above constructor must have been created by the corresponding version of `profile` or `cProfile`. To be specific, there is *no* file compatibility guaranteed with future versions of this profiler, and there is no compatibility with files produced by other profilers, or the same profiler run on a different operating system. If several files are provided, all the statistics for identical functions will be coalesced, so that an overall

view of several processes can be considered in a single report. If additional files need to be combined with data in an existing `Stats` object, the `add()` method can be used.

Instead of reading the profile data from a file, a `cProfile.Profile` or `profile.Profile` object can be used as the profile data source.

`Stats` objects have the following methods :

strip_dirs()

This method for the `Stats` class removes all leading path information from file names. It is very useful in reducing the size of the printout to fit within (close to) 80 columns. This method modifies the object, and the stripped information is lost. After performing a strip operation, the object is considered to have its entries in a "random" order, as it was just after object initialization and loading. If `strip_dirs()` causes two function names to be indistinguishable (they are on the same line of the same filename, and have the same function name), then the statistics for these two entries are accumulated into a single entry.

add(**filenames*)

This method of the `Stats` class accumulates additional profiling information into the current profiling object. Its arguments should refer to filenames created by the corresponding version of `profile.run()` or `cProfile.run()`. Statistics for identically named (re : file, line, name) functions are automatically accumulated into single function statistics.

dump_stats(*filename*)

Save the data loaded into the `Stats` object to a file named *filename*. The file is created if it does not exist, and is overwritten if it already exists. This is equivalent to the method of the same name on the `profile.Profile` and `cProfile.Profile` classes.

sort_stats(**keys*)

This method modifies the `Stats` object by sorting it according to the supplied criteria. The argument can be either a string or a SortKey enum identifying the basis of a sort (example : `'time'`, `'name'`, `SortKey.TIME` or `SortKey.NAME`). The SortKey enums argument have advantage over the string argument in that it is more robust and less error prone.

When more than one key is provided, then additional keys are used as secondary criteria when there is equality in all keys selected before them. For example, `sort_stats(SortKey.NAME, SortKey.FILE)` will sort all the entries according to their function name, and resolve all ties (identical function names) by sorting by file name.

For the string argument, abbreviations can be used for any key names, as long as the abbreviation is unambiguous.

The following are the valid string and SortKey :

Valid String Arg	Valid enum Arg	Signification
`'calls'`	SortKey.CALLS	call count
`'cumulative'`	SortKey.CUMULATIVE	cumulative time
`'cumtime'`	N/A	cumulative time
`'file'`	N/A	file name
`'filename'`	SortKey.FILENAME	file name
`'module'`	N/A	file name
`'ncalls'`	N/A	call count
`'pcalls'`	SortKey.PCALLS	primitive call count
`'line'`	SortKey.LINE	line number
`'name'`	SortKey.NAME	function name
`'nfl'`	SortKey.NFL	name/file/line
`'stdname'`	SortKey.STDNAME	standard name
`'time'`	SortKey.TIME	internal time
`'tottime'`	N/A	internal time

Note that all sorts on statistics are in descending order (placing most time consuming items first), where as name, file, and line number searches are in ascending order (alphabetical). The subtle distinction between

`SortKey.NFL` and `SortKey.STDNAME` is that the standard name is a sort of the name as printed, which means that the embedded line numbers get compared in an odd way. For example, lines 3, 20, and 40 would (if the file names were the same) appear in the string order 20, 3 and 40. In contrast, `SortKey.NFL` does a numeric compare of the line numbers. In fact, `sort_stats(SortKey.NFL)` is the same as `sort_stats(SortKey.NAME, SortKey.FILENAME, SortKey.LINE)`.

For backward-compatibility reasons, the numeric arguments `-1`, `0`, `1`, and `2` are permitted. They are interpreted as `'stdname'`, `'calls'`, `'time'`, and `'cumulative'` respectively. If this old style format (numeric) is used, only one sort key (the numeric key) will be used, and additional arguments will be silently ignored.

Ajouté dans la version 3.7 : Added the SortKey enum.

reverse_order()

This method for the *Stats* class reverses the ordering of the basic list within the object. Note that by default ascending vs descending order is properly selected based on the sort key of choice.

print_stats(**restrictions*)

This method for the *Stats* class prints out a report as described in the *profile.run()* definition.

The order of the printing is based on the last *sort_stats()* operation done on the object (subject to caveats in *add()* and *strip_dirs()*).

The arguments provided (if any) can be used to limit the list down to the significant entries. Initially, the list is taken to be the complete set of profiled functions. Each restriction is either an integer (to select a count of lines), or a decimal fraction between 0.0 and 1.0 inclusive (to select a percentage of lines), or a string that will interpreted as a regular expression (to pattern match the standard name that is printed). If several restrictions are provided, then they are applied sequentially. For example :

```
print_stats(.1, 'foo:')
```

would first limit the printing to first 10% of list, and then only print functions that were part of filename `.*foo:`. In contrast, the command :

```
print_stats('foo:', .1)
```

would limit the list to all functions having file names `.*foo:`, and then proceed to only print the first 10% of them.

print_callers(**restrictions*)

This method for the *Stats* class prints a list of all functions that called each function in the profiled database. The ordering is identical to that provided by *print_stats()*, and the definition of the restricting argument is also identical. Each caller is reported on its own line. The format differs slightly depending on the profiler that produced the stats :

— With *profile*, a number is shown in parentheses after each caller to show how many times this specific call was made. For convenience, a second non-parenthesized number repeats the cumulative time spent in the function at the right.

— With *cProfile*, each caller is preceded by three numbers : the number of times this specific call was made, and the total and cumulative times spent in the current function while it was invoked by this specific caller.

print_callees(**restrictions*)

This method for the *Stats* class prints a list of all function that were called by the indicated function. Aside from this reversal of direction of calls (re : called vs was called by), the arguments and ordering are identical to the *print_callers()* method.

get_stats_profile()

This method returns an instance of StatsProfile, which contains a mapping of function names to instances of FunctionProfile. Each FunctionProfile instance holds information related to the function's profile such as how long the function took to run, how many times it was called, etc...

Ajouté dans la version 3.9 : Added the following dataclasses : StatsProfile, FunctionProfile. Added the following function : get_stats_profile.

28.5.5 What Is Deterministic Profiling ?

Deterministic profiling is meant to reflect the fact that all *function call*, *function return*, and *exception* events are monitored, and precise timings are made for the intervals between these events (during which time the user's code is executing). In contrast, *statistical profiling* (which is not done by this module) randomly samples the effective instruction pointer, and deduces where time is being spent. The latter technique traditionally involves less overhead (as the code does not need to be instrumented), but provides only relative indications of where time is being spent.

In Python, since there is an interpreter active during execution, the presence of instrumented code is not required in order to do deterministic profiling. Python automatically provides a *hook* (optional callback) for each event. In addition, the interpreted nature of Python tends to add so much overhead to execution, that deterministic profiling tends to only add small processing overhead in typical applications. The result is that deterministic profiling is not that expensive, yet provides extensive run time statistics about the execution of a Python program.

Call count statistics can be used to identify bugs in code (surprising counts), and to identify possible inline-expansion points (high call counts). Internal time statistics can be used to identify "hot loops" that should be carefully optimized. Cumulative time statistics should be used to identify high level errors in the selection of algorithms. Note that the unusual handling of cumulative times in this profiler allows statistics for recursive implementations of algorithms to be directly compared to iterative implementations.

28.5.6 Limitations

One limitation has to do with accuracy of timing information. There is a fundamental problem with deterministic profilers involving accuracy. The most obvious restriction is that the underlying "clock" is only ticking at a rate (typically) of about .001 seconds. Hence no measurements will be more accurate than the underlying clock. If enough measurements are taken, then the "error" will tend to average out. Unfortunately, removing this first error induces a second source of error.

The second problem is that it "takes a while" from when an event is dispatched until the profiler's call to get the time actually *gets* the state of the clock. Similarly, there is a certain lag when exiting the profiler event handler from the time that the clock's value was obtained (and then squirreled away), until the user's code is once again executing. As a result, functions that are called many times, or call many functions, will typically accumulate this error. The error that accumulates in this fashion is typically less than the accuracy of the clock (less than one clock tick), but it *can* accumulate and become very significant.

The problem is more important with `profile` than with the lower-overhead `cProfile`. For this reason, `profile` provides a means of calibrating itself for a given platform so that this error can be probabilistically (on the average) removed. After the profiler is calibrated, it will be more accurate (in a least square sense), but it will sometimes produce negative numbers (when call counts are exceptionally low, and the gods of probability work against you :-).) Do *not* be alarmed by negative numbers in the profile. They should *only* appear if you have calibrated your profiler, and the results are actually better than without calibration.

28.5.7 Calibration

The profiler of the `profile` module subtracts a constant from each event handling time to compensate for the overhead of calling the time function, and socking away the results. By default, the constant is 0. The following procedure can be used to obtain a better constant for a given platform (see *Limitations*).

```
import profile
pr = profile.Profile()
for i in range(5):
    print(pr.calibrate(10000))
```

The method executes the number of Python calls given by the argument, directly and again under the profiler, measuring the time for both. It then computes the hidden overhead per profiler event, and returns that as a float. For example, on a 1.8Ghz Intel Core i5 running macOS, and using Python's time.process_time() as the timer, the magical number is about 4.04e-6.

The object of this exercise is to get a fairly consistent result. If your computer is *very* fast, or your timer function has poor resolution, you might have to pass 100000, or even 1000000, to get consistent results.

When you have a consistent answer, there are three ways you can use it :

```
import profile

# 1. Apply computed bias to all Profile instances created hereafter.
profile.Profile.bias = your_computed_bias

# 2. Apply computed bias to a specific Profile instance.
pr = profile.Profile()
pr.bias = your_computed_bias

# 3. Specify computed bias in instance constructor.
pr = profile.Profile(bias=your_computed_bias)
```

If you have a choice, you are better off choosing a smaller constant, and then your results will "less often" show up as negative in profile statistics.

28.5.8 Using a custom timer

If you want to change how current time is determined (for example, to force use of wall-clock time or elapsed process time), pass the timing function you want to the `Profile` class constructor :

```
pr = profile.Profile(your_time_func)
```

The resulting profiler will then call `your_time_func`. Depending on whether you are using *`profile.Profile`* or `cProfile.Profile`, `your_time_func`'s return value will be interpreted differently :

`profile.Profile`

`your_time_func` should return a single number, or a list of numbers whose sum is the current time (like what *`os.times()`* returns). If the function returns a single time number, or the list of returned numbers has length 2, then you will get an especially fast version of the dispatch routine.

Be warned that you should calibrate the profiler class for the timer function that you choose (see *Calibration*). For most machines, a timer that returns a lone integer value will provide the best results in terms of low overhead during profiling. (*`os.times()`* is *pretty* bad, as it returns a tuple of floating-point values). If you want to substitute a better timer in the cleanest fashion, derive a class and hardwire a replacement dispatch method that best handles your timer call, along with the appropriate calibration constant.

`cProfile.Profile`

`your_time_func` should return a single number. If it returns integers, you can also invoke the class constructor with a second argument specifying the real duration of one unit of time. For example, if `your_integer_time_func` returns times measured in thousands of seconds, you would construct the `Profile` instance as follows :

```
pr = cProfile.Profile(your_integer_time_func, 0.001)
```

As the `cProfile.Profile` class cannot be calibrated, custom timer functions should be used with care and should be as fast as possible. For the best results with a custom timer, it might be necessary to hard-code it in the C source of the internal `_lsprof` module.

Python 3.3 adds several new functions in *`time`* that can be used to make precise measurements of process or wall-clock time. For example, see *`time.perf_counter()`*.

28.6 `timeit` --- Measure execution time of small code snippets

Code source : Lib/timeit.py

Ce module fournit une façon simple de mesurer le temps d'exécution de fragments de code Python. Il expose une *Interface en ligne de commande* ainsi qu'une *interface Python*. Ce module permet d'éviter un certain nombre de problèmes classiques liés à la mesure des temps d'exécution. Voir par exemple à ce sujet l'introduction par Tim Peters du chapitre « Algorithmes » dans la seconde édition du livre *Python Cookbook*, aux éditions O'Reilly.

28.6.1 Exemples simples

L'exemple suivant illustre l'utilisation de l'*Interface en ligne de commande* afin de comparer trois expressions différentes :

```
$ python -m timeit "'-'.join(str(n) for n in range(100))"
10000 loops, best of 5: 30.2 usec per loop
$ python -m timeit "'-'.join([str(n) for n in range(100)])"
10000 loops, best of 5: 27.5 usec per loop
$ python -m timeit "'-'.join(map(str, range(100)))"
10000 loops, best of 5: 23.2 usec per loop
```

L'*Interface Python* peut être utilisée aux mêmes fins avec :

```
>>> import timeit
>>> timeit.timeit('"-".join(str(n) for n in range(100))', number=10000)
0.3018611848820001
>>> timeit.timeit('"-".join([str(n) for n in range(100)])', number=10000)
0.2727368790656328
>>> timeit.timeit('"-".join(map(str, range(100)))', number=10000)
0.23702679807320237
```

Un objet appelable peut également être passé en argument à l'*Interface Python* :

```
>>> timeit.timeit(lambda: "-".join(map(str, range(100))), number=10000)
0.19665591977536678
```

Notez cependant que `timeit()` détermine automatiquement le nombre de répétitions seulement lorsque l'interface en ligne de commande est utilisée. Vous pouvez trouver des exemples d'usages avancés dans la section *Exemples*.

28.6.2 Interface Python

Ce module définit une classe publique ainsi que trois fonctions destinées à simplifier son usage :

`timeit.`**`timeit`**(*stmt='pass'*, *setup='pass'*, *timer=<default timer>*, *number=1000000*, *globals=None*)

Crée une instance d'objet `Timer` à partir de l'instruction donnée, du code *setup* et de la fonction *timer*, puis exécute sa méthode `timeit()` à *number* reprises. L'argument optionnel *globals* spécifie un espace de nommage dans lequel exécuter le code.

Modifié dans la version 3.5 : Le paramètre optionnel *globals* a été ajouté.

`timeit.`**`repeat`**(*stmt='pass'*, *setup='pass'*, *timer=<default timer>*, *repeat=5*, *number=1000000*, *globals=None*)

Crée une instance d'objet `Timer` à partir de l'instruction donnée, du code *setup* et de la fonction *timer*, puis exécute sa méthode `repeat()` à *number* reprises, *repeat* fois. L'argument optionnel *globals* spécifie un espace de nommage dans lequel exécuter le code.

Modifié dans la version 3.5 : Le paramètre optionnel *globals* a été ajouté.

Modifié dans la version 3.7 : La valeur par défaut de *repeat* est passée de 3 à 5.

`timeit.`**`default_timer`**()

The default timer, which is always time.perf_counter(), returns float seconds. An alternative, time.perf_counter_ns, returns integer nanoseconds.

Modifié dans la version 3.3 : `time.perf_counter()` est désormais le minuteur par défaut.

`class` `timeit.`**`Timer`**(*stmt='pass'*, *setup='pass'*, *timer=<timer function>*, *globals=None*)

Classe permettant de mesurer le temps d'exécution de fragments de code.

Ce constructeur prend en argument une instruction dont le temps d'exécution doit être mesuré, une instruction additionnelle de mise en place et une fonction de chronométrage. Les deux instructions valent `'pass'` par défaut ; la fonction de chronométrage dépend de la plateforme d'exécution (se référer au *doc string* du module). *stmt* et *setup* peuvent contenir plusieurs instructions séparées par des `;` ou des sauts de lignes tant qu'ils ne

comportent pas de littéraux sur plusieurs lignes. L'instruction est exécutée dans l'espace de nommage de *timeit* par défaut ; ce comportement peut être modifié en passant un espace de nommage au paramètre *globals*.

Pour mesurer le temps d'exécution de la première instruction, utilisez la méthode `timeit()`. Les méthodes `repeat()` et `autorange()` sont des méthodes d'agrément permettant d'appeler `timeit()` à plusieurs reprises.

Le temps d'exécution de *setup* n'est pas pris en compte dans le temps global d'exécution.

Les paramètres *stmt* et *setup* peuvent également recevoir des objets appelables sans argument. Ceci transforme alors les appels à ces objets en fonction de chronométrage qui seront exécutées par `timeit()`. Notez que le surcoût lié à la mesure du temps d'exécution dans ce cas est légèrement supérieur en raisons des appels de fonction supplémentaires.

Modifié dans la version 3.5 : Le paramètre optionnel *globals* a été ajouté.

timeit(*number=1000000*)

Time *number* executions of the main statement. This executes the setup statement once, and then returns the time it takes to execute the main statement a number of times. The default timer returns seconds as a float. The argument is the number of times through the loop, defaulting to one million. The main statement, the setup statement and the timer function to be used are passed to the constructor.

> **Note**
>
> Par défaut, `timeit()` désactive temporairement le *ramasse-miettes* pendant le chronométrage. Cette approche a l'avantage de permettre de comparer des mesures indépendantes. L'inconvénient de cette méthode est que le ramasse-miettes peut avoir un impact significatif sur les performances de la fonction étudiée. Dans ce cas, le ramasse-miettes peut être réactivé en première instruction de la chaîne *setup*. Par exemple :
>
> ```
> timeit.Timer('for i in range(10): oct(i)', 'gc.enable()').timeit()
> ```

autorange(*callback=None*)

Détermine automatiquement combien de fois appeler `timeit()`.

This is a convenience function that calls `timeit()` repeatedly so that the total time >= 0.2 second, returning the eventual (number of loops, time taken for that number of loops). It calls `timeit()` with increasing numbers from the sequence 1, 2, 5, 10, 20, 50, ... until the time taken is at least 0.2 seconds.

Si *callback* est spécifié et n'est pas `None`, elle est appelée après chaque itération avec deux arguments (numéro de l'itération et temps écoulé) : `callback(number, time_taken)`.

Ajouté dans la version 3.6.

repeat(*repeat=5*, *number=1000000*)

Appelle `timeit()` plusieurs fois.

Cette fonction d'agrément appelle `timeit()` à plusieurs reprises et renvoie une liste de résultats. Le premier argument spécifie le nombre d'appels à `timeit()`. Le second argument spécifie l'argument *number* de `timeit()`.

> **Note**
>
> Il est tentant de vouloir calculer la moyenne et l'écart-type des résultats et notifier ces valeurs. Ce n'est cependant pas très utile. En pratique, la valeur la plus basse donne une estimation basse de la vitesse maximale à laquelle votre machine peut exécuter le fragment de code spécifié ; les valeurs hautes de la liste sont typiquement provoquées non pas par une variabilité de la vitesse d'exécution de Python, mais par d'autres processus interférant avec la précision du chronométrage. Le `min()` du résultat est probablement la seule valeur à laquelle vous devriez vous intéresser. Pour aller plus loin, vous devriez regarder l'intégralité des résultats et utiliser le bon sens plutôt que les statistiques.

Modifié dans la version 3.7 : La valeur par défaut de *repeat* est passée de 3 à 5.

print_exc(*file=None*)

Outil permettant d'afficher la trace du code chronométré.

Usage typique :

```
t = Timer(...)       # outside the try/except
try:
    t.timeit(...)    # or t.repeat(...)
except Exception:
    t.print_exc()
```

L'avantage par rapport à la trace standard est que les lignes sources du code compilé sont affichées. Le paramètre optionnel *file* définit l'endroit où la trace est envoyée, par défaut `sys.stderr`.

28.6.3 Interface en ligne de commande

Lorsque le module est appelé comme un programme en ligne de commande, la syntaxe suivante est utilisée :

```
python -m timeit [-n N] [-r N] [-u U] [-s S] [-p] [-v] [-h] [statement ...]
```

Les options suivantes sont gérées :

-n N, **--number**=N

nombre d'exécutions de l'instruction *statement*

-r N, **--repeat**=N

nombre de répétitions du chronomètre (5 par défaut)

-s S, **--setup**=S

instruction exécutée une seule fois à l'initialisation (`pass` par défaut)

-p, --process

mesure le temps au niveau du processus et non au niveau du système, en utilisant `time.process_time()` plutôt que `time.perf_counter()` qui est utilisée par défaut

Ajouté dans la version 3.3.

-u, --unit=U

specify a time unit for timer output ; can select `nsec`, `usec`, `msec`, or `sec`

Ajouté dans la version 3.5.

-v, --verbose

affiche les temps d'exécutions bruts, répéter pour plus de précision

-h, --help

affiche un court message d'aide puis quitte

Une instruction sur plusieurs lignes peut être donnée en entrée en spécifiant chaque ligne comme un argument séparé. Indenter une ligne est possible en encadrant l'argument de guillemets et en le préfixant par des espaces. Plusieurs `-s` sont gérées de la même façon.

Si `-n` n'est pas donnée, un nombre de boucles approprié est calculé en essayant des nombres croissants de la séquence 1, 2, 5, 10, 20, 50, ... jusqu'à ce que le temps total d'exécution dépasse 0,2 secondes.

Les mesures de `default_timer()` peuvent être altérées par d'autres programmes s'exécutant sur la même machine. La meilleure approche lorsqu'un chronométrage exact est nécessaire est de répéter celui-ci à plusieurs reprises et considérer le meilleur temps. L'option `-r` est adaptée à ce fonctionnement, les cinq répétitions par défaut suffisent probablement dans la plupart des cas. Vous pouvez utiliser `time.process_time()` pour mesurer le temps processeur.

> **Note**
>
> Il existe un surcoût minimal associé à l'exécution de l'instruction `pass`. Le code présenté ici ne tente pas de le masquer, mais vous devez être conscient de son existence. Ce surcoût minimal peut être mesuré en invoquant le programme sans argument ; il peut différer en fonction des versions de Python.

28.6.4 Exemples

Il est possible de fournir une instruction de mise en place exécutée une seule fois au début du chronométrage :

```
$ python -m timeit -s "text = 'sample string'; char = 'g'" "char in text"
5000000 loops, best of 5: 0.0877 usec per loop
$ python -m timeit -s "text = 'sample string'; char = 'g'" "text.find(char)"
1000000 loops, best of 5: 0.342 usec per loop
```

In the output, there are three fields. The loop count, which tells you how many times the statement body was run per timing loop repetition. The repetition count ('best of 5') which tells you how many times the timing loop was repeated, and finally the time the statement body took on average within the best repetition of the timing loop. That is, the time the fastest repetition took divided by the loop count.

```
>>> import timeit
>>> timeit.timeit('char in text', setup='text = "sample string"; char = "g"')
0.41440500499993504
>>> timeit.timeit('text.find(char)', setup='text = "sample string"; char = "g"')
1.7246671520006203
```

La même chose peut être réalisée en utilisant la classe `Timer` et ses méthodes :

```
>>> import timeit
>>> t = timeit.Timer('char in text', setup='text = "sample string"; char = "g"')
>>> t.timeit()
0.3955516149999312
>>> t.repeat()
[0.40183617287970225, 0.37027556854118704, 0.38344867356679524, 0.3712595970846668,
↪ 0.37866875250654886]
```

Les exemples qui suivent montrent comment chronométrer des expressions sur plusieurs lignes. Nous comparons ici le coût d'utilisation de `hasattr()` par rapport à `try/except` pour tester la présence ou l'absence d'attributs d'un objet :

```
$ python -m timeit "try:" "  str.__bool__" "except AttributeError:" "  pass"
20000 loops, best of 5: 15.7 usec per loop
$ python -m timeit "if hasattr(str, '__bool__'): pass"
50000 loops, best of 5: 4.26 usec per loop

$ python -m timeit "try:" "  int.__bool__" "except AttributeError:" "  pass"
200000 loops, best of 5: 1.43 usec per loop
$ python -m timeit "if hasattr(int, '__bool__'): pass"
100000 loops, best of 5: 2.23 usec per loop
```

```
>>> import timeit
>>> # attribute is missing
>>> s = """\
... try:
...     str.__bool__
... except AttributeError:
...     pass
```

(suite sur la page suivante)

(suite de la page précédente)

```
... """
>>> timeit.timeit(stmt=s, number=100000)
0.9138244460009446
>>> s = "if hasattr(str, '__bool__'): pass"
>>> timeit.timeit(stmt=s, number=100000)
0.5829014980008651
>>>
>>> # attribute is present
>>> s = """\
... try:
...     int.__bool__
... except AttributeError:
...     pass
... """
>>> timeit.timeit(stmt=s, number=100000)
0.04215312199994514
>>> s = "if hasattr(int, '__bool__'): pass"
>>> timeit.timeit(stmt=s, number=100000)
0.08588060699912603
```

Afin de permettre à `timeit` d'accéder aux fonctions que vous avez définies, vous pouvez passer au paramètre *setup* une instruction d'importation :

```
def test():
    """Stupid test function"""
    L = [i for i in range(100)]

if __name__ == '__main__':
    import timeit
    print(timeit.timeit("test()", setup="from __main__ import test"))
```

Une autre possibilité est de passer `globals()` au paramètre *globals*, ceci qui exécutera le code dans l'espace de nommage global courant. Cela peut être plus pratique que de spécifier manuellement des importations :

```
def f(x):
    return x**2
def g(x):
    return x**4
def h(x):
    return x**8

import timeit
print(timeit.timeit('[func(42) for func in (f,g,h)]', globals=globals()))
```

28.7 `trace` --- Trace or track Python statement execution

Code source : Lib/abc.py

The `trace` module allows you to trace program execution, generate annotated statement coverage listings, print caller/callee relationships and list functions executed during a program run. It can be used in another program or from the command line.

Voir aussi

Coverage.py
A popular third-party coverage tool that provides HTML output along with advanced features such as branch coverage.

28.7.1 Utilisation en ligne de commande.

The *trace* module can be invoked from the command line. It can be as simple as

```
python -m trace --count -C . somefile.py ...
```

The above will execute `somefile.py` and generate annotated listings of all Python modules imported during the execution into the current directory.

--help
Display usage and exit.

--version
Display the version of the module and exit.

Ajouté dans la version 3.8 : Added `--module` option that allows to run an executable module.

Main options

At least one of the following options must be specified when invoking *trace*. The *--listfuncs* option is mutually exclusive with the *--trace* and *--count* options. When *--listfuncs* is provided, neither *--count* nor *--trace* are accepted, and vice versa.

-c, --count
Produce a set of annotated listing files upon program completion that shows how many times each statement was executed. See also *--coverdir*, *--file* and *--no-report* below.

-t, --trace
Display lines as they are executed.

-l, --listfuncs
Display the functions executed by running the program.

-r, --report
Produce an annotated list from an earlier program run that used the *--count* and *--file* option. This does not execute any code.

-T, --trackcalls
Display the calling relationships exposed by running the program.

Modifiers

-f, --file=<file>
Name of a file to accumulate counts over several tracing runs. Should be used with the *--count* option.

-C, --coverdir=<dir>
Directory where the report files go. The coverage report for `package.module` is written to file *dir/package/module*`.cover`.

-m, --missing
When generating annotated listings, mark lines which were not executed with `>>>>>>`.

-s, --summary
When using *--count* or *--report*, write a brief summary to stdout for each file processed.

-R, --no-report

Do not generate annotated listings. This is useful if you intend to make several runs with `--count`, and then produce a single set of annotated listings at the end.

-g, --timing

Prefix each line with the time since the program started. Only used while tracing.

Filters

These options may be repeated multiple times.

--ignore-module=<mod>

Ignore each of the given module names and its submodules (if it is a package). The argument can be a list of names separated by a comma.

--ignore-dir=<dir>

Ignore all modules and packages in the named directory and subdirectories. The argument can be a list of directories separated by `os.pathsep`.

28.7.2 Programmatic Interface

class trace.**Trace**(*count=1*, *trace=1*, *countfuncs=0*, *countcallers=0*, *ignoremods=()*, *ignoredirs=()*, *infile=None*, *outfile=None*, *timing=False*)

Create an object to trace execution of a single statement or expression. All parameters are optional. *count* enables counting of line numbers. *trace* enables line execution tracing. *countfuncs* enables listing of the functions called during the run. *countcallers* enables call relationship tracking. *ignoremods* is a list of modules or packages to ignore. *ignoredirs* is a list of directories whose modules or packages should be ignored. *infile* is the name of the file from which to read stored count information. *outfile* is the name of the file in which to write updated count information. *timing* enables a timestamp relative to when tracing was started to be displayed.

run(*cmd*)

Execute the command and gather statistics from the execution with the current tracing parameters. *cmd* must be a string or code object, suitable for passing into `exec()`.

runctx(*cmd*, *globals=None*, *locals=None*)

Execute the command and gather statistics from the execution with the current tracing parameters, in the defined global and local environments. If not defined, *globals* and *locals* default to empty dictionaries.

runfunc(*func*, */*, **args*, ***kwds*)

Call *func* with the given arguments under control of the `Trace` object with the current tracing parameters.

results()

Return a `CoverageResults` object that contains the cumulative results of all previous calls to `run`, `runctx` and `runfunc` for the given `Trace` instance. Does not reset the accumulated trace results.

class trace.**CoverageResults**

A container for coverage results, created by `Trace.results()`. Should not be created directly by the user.

update(*other*)

Merge in data from another `CoverageResults` object.

write_results(*show_missing=True*, *summary=False*, *coverdir=None*, ***, *ignore_missing_files=False*)

Write coverage results. Set *show_missing* to show lines that had no hits. Set *summary* to include in the output the coverage summary per module. *coverdir* specifies the directory into which the coverage result files will be output. If `None`, the results for each source file are placed in its directory.

If *ignore_missing_files* is `True`, coverage counts for files that no longer exist are silently ignored. Otherwise, a missing file will raise a `FileNotFoundError`.

Modifié dans la version 3.13 : Added *ignore_missing_files* parameter.

A simple example demonstrating the use of the programmatic interface :

```
import sys
import trace

# create a Trace object, telling it what to ignore, and whether to
# do tracing or line-counting or both.
tracer = trace.Trace(
    ignoredirs=[sys.prefix, sys.exec_prefix],
    trace=0,
    count=1)

# run the new command using the given tracer
tracer.run('main()')

# make a report, placing output in the current directory
r = tracer.results()
r.write_results(show_missing=True, coverdir=".")
```

28.8 `tracemalloc` --- Trace memory allocations

Ajouté dans la version 3.4.

Source code : Lib/tracemalloc.py

The tracemalloc module is a debug tool to trace memory blocks allocated by Python. It provides the following information :

- Traceback where an object was allocated
- Statistics on allocated memory blocks per filename and per line number : total size, number and average size of allocated memory blocks
- Compute the differences between two snapshots to detect memory leaks

To trace most memory blocks allocated by Python, the module should be started as early as possible by setting the `PYTHONTRACEMALLOC` environment variable to `1`, or by using `-X tracemalloc` command line option. The *`tracemalloc.start()`* function can be called at runtime to start tracing Python memory allocations.

By default, a trace of an allocated memory block only stores the most recent frame (1 frame). To store 25 frames at startup : set the `PYTHONTRACEMALLOC` environment variable to `25`, or use the `-X tracemalloc=25` command line option.

28.8.1 Exemples

Display the top 10

Display the 10 files allocating the most memory :

```
import tracemalloc

tracemalloc.start()

# ... run your application ...

snapshot = tracemalloc.take_snapshot()
top_stats = snapshot.statistics('lineno')

print("[ Top 10 ]")
for stat in top_stats[:10]:
    print(stat)
```

Example of output of the Python test suite :

```
[ Top 10 ]
<frozen importlib._bootstrap>:716: size=4855 KiB, count=39328, average=126 B
<frozen importlib._bootstrap>:284: size=521 KiB, count=3199, average=167 B
/usr/lib/python3.4/collections/__init__.py:368: size=244 KiB, count=2315,␣
↪average=108 B
/usr/lib/python3.4/unittest/case.py:381: size=185 KiB, count=779, average=243 B
/usr/lib/python3.4/unittest/case.py:402: size=154 KiB, count=378, average=416 B
/usr/lib/python3.4/abc.py:133: size=88.7 KiB, count=347, average=262 B
<frozen importlib._bootstrap>:1446: size=70.4 KiB, count=911, average=79 B
<frozen importlib._bootstrap>:1454: size=52.0 KiB, count=25, average=2131 B
<string>:5: size=49.7 KiB, count=148, average=344 B
/usr/lib/python3.4/sysconfig.py:411: size=48.0 KiB, count=1, average=48.0 KiB
```

We can see that Python loaded `4855 KiB` data (bytecode and constants) from modules and that the `collections` module allocated `244 KiB` to build `namedtuple` types.

See `Snapshot.statistics()` for more options.

Compute differences

Take two snapshots and display the differences :

```
import tracemalloc
tracemalloc.start()
# ... start your application ...

snapshot1 = tracemalloc.take_snapshot()
# ... call the function leaking memory ...
snapshot2 = tracemalloc.take_snapshot()

top_stats = snapshot2.compare_to(snapshot1, 'lineno')

print("[ Top 10 differences ]")
for stat in top_stats[:10]:
    print(stat)
```

Example of output before/after running some tests of the Python test suite :

```
[ Top 10 differences ]
<frozen importlib._bootstrap>:716: size=8173 KiB (+4428 KiB), count=71332 (+39369),
↪ average=117 B
/usr/lib/python3.4/linecache.py:127: size=940 KiB (+940 KiB), count=8106 (+8106),␣
↪average=119 B
/usr/lib/python3.4/unittest/case.py:571: size=298 KiB (+298 KiB), count=589 (+589),
↪ average=519 B
<frozen importlib._bootstrap>:284: size=1005 KiB (+166 KiB), count=7423 (+1526),␣
↪average=139 B
/usr/lib/python3.4/mimetypes.py:217: size=112 KiB (+112 KiB), count=1334 (+1334),␣
↪average=86 B
/usr/lib/python3.4/http/server.py:848: size=96.0 KiB (+96.0 KiB), count=1 (+1),␣
↪average=96.0 KiB
/usr/lib/python3.4/inspect.py:1465: size=83.5 KiB (+83.5 KiB), count=109 (+109),␣
↪average=784 B
/usr/lib/python3.4/unittest/mock.py:491: size=77.7 KiB (+77.7 KiB), count=143␣
↪(+143), average=557 B
/usr/lib/python3.4/urllib/parse.py:476: size=71.8 KiB (+71.8 KiB), count=969␣
↪(+969), average=76 B
/usr/lib/python3.4/contextlib.py:38: size=67.2 KiB (+67.2 KiB), count=126 (+126),␣
```

(suite sur la page suivante)

(suite de la page précédente)

```
↪average=546 B
```

We can see that Python has loaded `8173 KiB` of module data (bytecode and constants), and that this is `4428 KiB` more than had been loaded before the tests, when the previous snapshot was taken. Similarly, the `linecache` module has cached `940 KiB` of Python source code to format tracebacks, all of it since the previous snapshot.

If the system has little free memory, snapshots can be written on disk using the `Snapshot.dump()` method to analyze the snapshot offline. Then use the `Snapshot.load()` method reload the snapshot.

Get the traceback of a memory block

Code to display the traceback of the biggest memory block :

```
import tracemalloc

# Store 25 frames
tracemalloc.start(25)

# ... run your application ...

snapshot = tracemalloc.take_snapshot()
top_stats = snapshot.statistics('traceback')

# pick the biggest memory block
stat = top_stats[0]
print("%s memory blocks: %.1f KiB" % (stat.count, stat.size / 1024))
for line in stat.traceback.format():
    print(line)
```

Example of output of the Python test suite (traceback limited to 25 frames) :

```
903 memory blocks: 870.1 KiB
  File "<frozen importlib._bootstrap>", line 716
  File "<frozen importlib._bootstrap>", line 1036
  File "<frozen importlib._bootstrap>", line 934
  File "<frozen importlib._bootstrap>", line 1068
  File "<frozen importlib._bootstrap>", line 619
  File "<frozen importlib._bootstrap>", line 1581
  File "<frozen importlib._bootstrap>", line 1614
  File "/usr/lib/python3.4/doctest.py", line 101
    import pdb
  File "<frozen importlib._bootstrap>", line 284
  File "<frozen importlib._bootstrap>", line 938
  File "<frozen importlib._bootstrap>", line 1068
  File "<frozen importlib._bootstrap>", line 619
  File "<frozen importlib._bootstrap>", line 1581
  File "<frozen importlib._bootstrap>", line 1614
  File "/usr/lib/python3.4/test/support/__init__.py", line 1728
    import doctest
  File "/usr/lib/python3.4/test/test_pickletools.py", line 21
    support.run_doctest(pickletools)
  File "/usr/lib/python3.4/test/regrtest.py", line 1276
    test_runner()
  File "/usr/lib/python3.4/test/regrtest.py", line 976
    display_failure=not verbose)
  File "/usr/lib/python3.4/test/regrtest.py", line 761
    match_tests=ns.match_tests)
```

(suite sur la page suivante)

(suite de la page précédente)

```
  File "/usr/lib/python3.4/test/regrtest.py", line 1563
    main()
  File "/usr/lib/python3.4/test/__main__.py", line 3
    regrtest.main_in_temp_cwd()
  File "/usr/lib/python3.4/runpy.py", line 73
    exec(code, run_globals)
  File "/usr/lib/python3.4/runpy.py", line 160
    "__main__", fname, loader, pkg_name)
```

We can see that the most memory was allocated in the *importlib* module to load data (bytecode and constants) from modules : `870.1 KiB`. The traceback is where the *importlib* loaded data most recently : on the `import pdb` line of the *doctest* module. The traceback may change if a new module is loaded.

Pretty top

Code to display the 10 lines allocating the most memory with a pretty output, ignoring `<frozen importlib._bootstrap>` and `<unknown>` files :

```
import linecache
import os
import tracemalloc

def display_top(snapshot, key_type='lineno', limit=10):
    snapshot = snapshot.filter_traces((
        tracemalloc.Filter(False, "<frozen importlib._bootstrap>"),
        tracemalloc.Filter(False, "<unknown>"),
    ))
    top_stats = snapshot.statistics(key_type)

    print("Top %s lines" % limit)
    for index, stat in enumerate(top_stats[:limit], 1):
        frame = stat.traceback[0]
        print("#%s: %s:%s: %.1f KiB"
              % (index, frame.filename, frame.lineno, stat.size / 1024))
        line = linecache.getline(frame.filename, frame.lineno).strip()
        if line:
            print('    %s' % line)

    other = top_stats[limit:]
    if other:
        size = sum(stat.size for stat in other)
        print("%s other: %.1f KiB" % (len(other), size / 1024))
    total = sum(stat.size for stat in top_stats)
    print("Total allocated size: %.1f KiB" % (total / 1024))

tracemalloc.start()

# ... run your application ...

snapshot = tracemalloc.take_snapshot()
display_top(snapshot)
```

Example of output of the Python test suite :

```
Top 10 lines
#1: Lib/base64.py:414: 419.8 KiB
    _b85chars2 = [(a + b) for a in _b85chars for b in _b85chars]
```

(suite sur la page suivante)

(suite de la page précédente)

```
#2: Lib/base64.py:306: 419.8 KiB
    _a85chars2 = [(a + b) for a in _a85chars for b in _a85chars]
#3: collections/__init__.py:368: 293.6 KiB
    exec(class_definition, namespace)
#4: Lib/abc.py:133: 115.2 KiB
    cls = super().__new__(mcls, name, bases, namespace)
#5: unittest/case.py:574: 103.1 KiB
    testMethod()
#6: Lib/linecache.py:127: 95.4 KiB
    lines = fp.readlines()
#7: urllib/parse.py:476: 71.8 KiB
    for a in _hexdig for b in _hexdig}
#8: <string>:5: 62.0 KiB
#9: Lib/_weakrefset.py:37: 60.0 KiB
    self.data = set()
#10: Lib/base64.py:142: 59.8 KiB
    _b32tab2 = [a + b for a in _b32tab for b in _b32tab]
6220 other: 3602.8 KiB
Total allocated size: 5303.1 KiB
```

See `Snapshot.statistics()` for more options.

Record the current and peak size of all traced memory blocks

The following code computes two sums like `0 + 1 + 2 + ...` inefficiently, by creating a list of those numbers. This list consumes a lot of memory temporarily. We can use `get_traced_memory()` and `reset_peak()` to observe the small memory usage after the sum is computed as well as the peak memory usage during the computations :

```
import tracemalloc

tracemalloc.start()

# Example code: compute a sum with a large temporary list
large_sum = sum(list(range(100000)))

first_size, first_peak = tracemalloc.get_traced_memory()

tracemalloc.reset_peak()

# Example code: compute a sum with a small temporary list
small_sum = sum(list(range(1000)))

second_size, second_peak = tracemalloc.get_traced_memory()

print(f"{first_size=}, {first_peak=}")
print(f"{second_size=}, {second_peak=}")
```

Sortie :

```
first_size=664, first_peak=3592984
second_size=804, second_peak=29704
```

Using `reset_peak()` ensured we could accurately record the peak during the computation of `small_sum`, even though it is much smaller than the overall peak size of memory blocks since the `start()` call. Without the call to `reset_peak()`, `second_peak` would still be the peak from the computation `large_sum` (that is, equal to `first_peak`). In this case, both peaks are much higher than the final memory usage, and which suggests we could optimise (by removing the unnecessary call to `list`, and writing `sum(range(...))`).

28.8.2 API

Fonctions

`tracemalloc.clear_traces()`

Clear traces of memory blocks allocated by Python.

See also `stop()`.

`tracemalloc.get_object_traceback(obj)`

Get the traceback where the Python object *obj* was allocated. Return a `Traceback` instance, or `None` if the `tracemalloc` module is not tracing memory allocations or did not trace the allocation of the object.

See also `gc.get_referrers()` and `sys.getsizeof()` functions.

`tracemalloc.get_traceback_limit()`

Get the maximum number of frames stored in the traceback of a trace.

The `tracemalloc` module must be tracing memory allocations to get the limit, otherwise an exception is raised.

The limit is set by the `start()` function.

`tracemalloc.get_traced_memory()`

Get the current size and peak size of memory blocks traced by the `tracemalloc` module as a tuple : `(current: int, peak: int)`.

`tracemalloc.reset_peak()`

Set the peak size of memory blocks traced by the `tracemalloc` module to the current size.

Do nothing if the `tracemalloc` module is not tracing memory allocations.

This function only modifies the recorded peak size, and does not modify or clear any traces, unlike `clear_traces()`. Snapshots taken with `take_snapshot()` before a call to `reset_peak()` can be meaningfully compared to snapshots taken after the call.

See also `get_traced_memory()`.

Ajouté dans la version 3.9.

`tracemalloc.get_tracemalloc_memory()`

Get the memory usage in bytes of the `tracemalloc` module used to store traces of memory blocks. Return an `int`.

`tracemalloc.is_tracing()`

`True` if the `tracemalloc` module is tracing Python memory allocations, `False` otherwise.

See also `start()` and `stop()` functions.

`tracemalloc.start(nframe : int = 1)`

Start tracing Python memory allocations : install hooks on Python memory allocators. Collected tracebacks of traces will be limited to *nframe* frames. By default, a trace of a memory block only stores the most recent frame : the limit is `1`. *nframe* must be greater or equal to `1`.

You can still read the original number of total frames that composed the traceback by looking at the `Traceback.total_nframe` attribute.

Storing more than `1` frame is only useful to compute statistics grouped by `'traceback'` or to compute cumulative statistics : see the `Snapshot.compare_to()` and `Snapshot.statistics()` methods.

Storing more frames increases the memory and CPU overhead of the `tracemalloc` module. Use the `get_tracemalloc_memory()` function to measure how much memory is used by the `tracemalloc` module.

The `PYTHONTRACEMALLOC` environment variable (`PYTHONTRACEMALLOC=NFRAME`) and the `-X tracemalloc=NFRAME` command line option can be used to start tracing at startup.

See also `stop()`, `is_tracing()` and `get_traceback_limit()` functions.

`tracemalloc.stop()`

Stop tracing Python memory allocations : uninstall hooks on Python memory allocators. Also clears all previously collected traces of memory blocks allocated by Python.

Call `take_snapshot()` function to take a snapshot of traces before clearing them.

See also `start()`, `is_tracing()` and `clear_traces()` functions.

`tracemalloc.take_snapshot()`

Take a snapshot of traces of memory blocks allocated by Python. Return a new `Snapshot` instance.

The snapshot does not include memory blocks allocated before the `tracemalloc` module started to trace memory allocations.

Tracebacks of traces are limited to `get_traceback_limit()` frames. Use the *nframe* parameter of the `start()` function to store more frames.

The `tracemalloc` module must be tracing memory allocations to take a snapshot, see the `start()` function.

See also the `get_object_traceback()` function.

DomainFilter

class `tracemalloc.DomainFilter`(*inclusive :* bool, *domain :* int)

Filter traces of memory blocks by their address space (domain).

Ajouté dans la version 3.6.

inclusive

If *inclusive* is `True` (include), match memory blocks allocated in the address space `domain`.

If *inclusive* is `False` (exclude), match memory blocks not allocated in the address space `domain`.

domain

Address space of a memory block (`int`). Read-only property.

Filter

class `tracemalloc.Filter`(*inclusive :* bool, *filename_pattern :* str, *lineno :* int = *None*, *all_frames :* bool = *False*, *domain :* int = *None*)

Filter on traces of memory blocks.

See the `fnmatch.fnmatch()` function for the syntax of *filename_pattern*. The `'.pyc'` file extension is replaced with `'.py'`.

Exemples :
— `Filter(True, subprocess.__file__)` only includes traces of the `subprocess` module
— `Filter(False, tracemalloc.__file__)` excludes traces of the `tracemalloc` module
— `Filter(False, "<unknown>")` excludes empty tracebacks
Modifié dans la version 3.5 : The `'.pyo'` file extension is no longer replaced with `'.py'`.

Modifié dans la version 3.6 : Added the `domain` attribute.

domain

Address space of a memory block (`int` or `None`).

tracemalloc uses the domain `0` to trace memory allocations made by Python. C extensions can use other domains to trace other resources.

inclusive

If *inclusive* is `True` (include), only match memory blocks allocated in a file with a name matching `filename_pattern` at line number `lineno`.

If *inclusive* is `False` (exclude), ignore memory blocks allocated in a file with a name matching `filename_pattern` at line number `lineno`.

lineno

Line number (`int`) of the filter. If *lineno* is `None`, the filter matches any line number.

filename_pattern

Filename pattern of the filter (`str`). Read-only property.

all_frames

If *all_frames* is `True`, all frames of the traceback are checked. If *all_frames* is `False`, only the most recent frame is checked.

This attribute has no effect if the traceback limit is `1`. See the `get_traceback_limit()` function and `Snapshot.traceback_limit` attribute.

Frame

class tracemalloc.**Frame**

Frame of a traceback.

The `Traceback` class is a sequence of `Frame` instances.

filename

Filename (`str`).

lineno

Line number (`int`).

Snapshot

class tracemalloc.**Snapshot**

Snapshot of traces of memory blocks allocated by Python.

The `take_snapshot()` function creates a snapshot instance.

compare_to(*old_snapshot :* Snapshot, *key_type :* str, *cumulative :* bool = *False*)

Compute the differences with an old snapshot. Get statistics as a sorted list of `StatisticDiff` instances grouped by *key_type*.

See the `Snapshot.statistics()` method for *key_type* and *cumulative* parameters.

The result is sorted from the biggest to the smallest by : absolute value of `StatisticDiff.size_diff`, `StatisticDiff.size`, absolute value of `StatisticDiff.count_diff`, `Statistic.count` and then by `StatisticDiff.traceback`.

dump(*filename*)

Write the snapshot into a file.

Use `load()` to reload the snapshot.

filter_traces(*filters*)

Create a new `Snapshot` instance with a filtered `traces` sequence, *filters* is a list of `DomainFilter` and `Filter` instances. If *filters* is an empty list, return a new `Snapshot` instance with a copy of the traces.

All inclusive filters are applied at once, a trace is ignored if no inclusive filters match it. A trace is ignored if at least one exclusive filter matches it.

Modifié dans la version 3.6 : `DomainFilter` instances are now also accepted in *filters*.

classmethod load(*filename*)

Load a snapshot from a file.

See also `dump()`.

statistics(*key_type :* str, *cumulative :* bool = *False*)

Get statistics as a sorted list of `Statistic` instances grouped by *key_type* :

key_type	description
`'filename'`	filename
`'lineno'`	filename and line number
`'traceback'`	traceback

If *cumulative* is `True`, cumulate size and count of memory blocks of all frames of the traceback of a trace, not only the most recent frame. The cumulative mode can only be used with *key_type* equals to `'filename'` and `'lineno'`.

The result is sorted from the biggest to the smallest by : `Statistic.size`, `Statistic.count` and then by `Statistic.traceback`.

traceback_limit

Maximum number of frames stored in the traceback of `traces` : result of the `get_traceback_limit()` when the snapshot was taken.

traces

Traces of all memory blocks allocated by Python : sequence of `Trace` instances.

The sequence has an undefined order. Use the `Snapshot.statistics()` method to get a sorted list of statistics.

Statistic

class tracemalloc.**Statistic**

Statistic on memory allocations.

`Snapshot.statistics()` returns a list of `Statistic` instances.

See also the `StatisticDiff` class.

count

Number of memory blocks (`int`).

size

Total size of memory blocks in bytes (`int`).

traceback

Traceback where the memory block was allocated, `Traceback` instance.

StatisticDiff

class tracemalloc.**StatisticDiff**

Statistic difference on memory allocations between an old and a new `Snapshot` instance.

`Snapshot.compare_to()` returns a list of `StatisticDiff` instances. See also the `Statistic` class.

count

Number of memory blocks in the new snapshot (`int`) : `0` if the memory blocks have been released in the new snapshot.

count_diff

Difference of number of memory blocks between the old and the new snapshots (`int`) : `0` if the memory blocks have been allocated in the new snapshot.

size

Total size of memory blocks in bytes in the new snapshot (`int`) : `0` if the memory blocks have been released in the new snapshot.

size_diff

Difference of total size of memory blocks in bytes between the old and the new snapshots (`int`) : `0` if the memory blocks have been allocated in the new snapshot.

traceback

Traceback where the memory blocks were allocated, `Traceback` instance.

Trace

class tracemalloc.**Trace**

Trace of a memory block.

The `Snapshot.traces` attribute is a sequence of `Trace` instances.

Modifié dans la version 3.6 : Added the `domain` attribute.

domain

Address space of a memory block (`int`). Read-only property.

tracemalloc uses the domain `0` to trace memory allocations made by Python. C extensions can use other domains to trace other resources.

size

Size of the memory block in bytes (`int`).

traceback

Traceback where the memory block was allocated, `Traceback` instance.

Traceback

class tracemalloc.**Traceback**

Sequence of `Frame` instances sorted from the oldest frame to the most recent frame.

A traceback contains at least `1` frame. If the `tracemalloc` module failed to get a frame, the filename `"<unknown>"` at line number `0` is used.

When a snapshot is taken, tracebacks of traces are limited to `get_traceback_limit()` frames. See the `take_snapshot()` function. The original number of frames of the traceback is stored in the `Traceback.total_nframe` attribute. That allows to know if a traceback has been truncated by the traceback limit.

The `Trace.traceback` attribute is an instance of `Traceback` instance.

Modifié dans la version 3.7 : Frames are now sorted from the oldest to the most recent, instead of most recent to oldest.

total_nframe

Total number of frames that composed the traceback before truncation. This attribute can be set to `None` if the information is not available.

Modifié dans la version 3.9 : The `Traceback.total_nframe` attribute was added.

format(*limit=None*, *most_recent_first=False*)

Format the traceback as a list of lines. Use the `linecache` module to retrieve lines from the source code. If *limit* is set, format the *limit* most recent frames if *limit* is positive. Otherwise, format the `abs(limit)` oldest frames. If *most_recent_first* is `True`, the order of the formatted frames is reversed, returning the most recent frame first instead of last.

Similar to the `traceback.format_tb()` function, except that `format()` does not include newlines.

Exemple :

```
print("Traceback (most recent call first):")
for line in traceback:
    print(line)
```

Sortie :

```
Traceback (most recent call first):
  File "test.py", line 9
    obj = Object()
  File "test.py", line 12
    tb = tracemalloc.get_object_traceback(f())
```

CHAPITRE 29

Paquets et distribution de paquets logiciels

Ces bibliothèques vous aident lors de la publication et l'installation de logiciels Python. Bien que ces modules sont conçus pour fonctionner avec le Python Package Index, ils peuvent aussi être utilisés avec un serveur local, ou sans serveur.

29.1 `ensurepip` --- Bootstrapping the `pip` installer

Ajouté dans la version 3.4.

Source code : Lib/ensurepip

Le module `ensurepip` met en place l'installateur `pip` dans un environnement Python, classique ou virtuel. Ce principe d'amorçage a été choisi car `pip` est un projet séparé de Python avec son propre cycle de versions. Il permet en particulier d'embarquer la version la plus récente de `pip` dans les mises à jour de maintenance de l'interpréteur CPython comme dans les nouvelles versions principales.

Dans la plupart des cas, il n'est pas nécessaire de recourir à ce module. `pip` est le plus souvent déjà installé pour vous. Cependant, `ensurepip` peut s'avérer utile si l'installation de `pip` a été sautée au moment de l'installation de Python (ou en créant un environnement virtuel), ou bien si `pip` a été désinstallé par l'utilisateur.

Note

Ce module n'accède *pas* au réseau. Tout ce qu'il faut pour amorcer `pip` est compris dans le paquet.

Voir aussi

installing-index
Guide de l'utilisateur final pour installer des paquets Python

PEP 453 : Amorçage explicite de pip dans les installations de Python
Les motivations pour l'ajout de ce module et sa spécification d'origine

Availability : not Android, not iOS, not WASI.

This module is not supported on *mobile platforms* or *WebAssembly platforms*.

29.1.1 Interface en ligne de commande

On fait appel à l'interface en ligne de commande à l'aide de l'option `-m` de l'interpréteur.

L'invocation la plus simple est :

```
python -m ensurepip
```

Cette commande installe `pip` s'il n'est pas déjà présent. Sinon, elle ne fait rien. Pour s'assurer que la version de `pip` est au moins aussi récente que celle embarquée dans `ensurepip`, passer l'option `--upgrade` :

```
python -m ensurepip --upgrade
```

`pip` est installé par défaut dans l'environnement virtuel courant, s'il y en a un, ou bien dans le dossier `site-packages` du système. L'emplacement d'installation se règle à travers deux options :

- `--root dir` : Installs `pip` relative to the given root directory rather than the root of the currently active virtual environment (if any) or the default root for the current Python installation.
- `--user` : installe `pip` dans le dossier `site-packages` propre à l'utilisateur au lieu du dossier global de l'installation de Python. Cette option n'est pas valide dans un environnement virtuel.

Par défaut, les commandes `pipX` et `pipX.Y` sont créées (où X.Y est la version de Python avec laquelle `ensurepip` est utilisé). Cela se contrôle par deux options supplémentaires :

- `--altinstall` : dans ce mode d'« installation parallèle », seule la commande `pipX.Y` est ajoutée, et pas la commande `pipX`.
- `--default-pip` : ce mode d'« installation de la version par défaut » crée la commande `pip` en plus de `pipX` et `pipX.Y`.

Combiner ces deux options conduit à une exception.

29.1.2 API du module

Le module *`ensurepip`* définit deux fonctions pour utilisation dans les programmes :

`ensurepip.`**`version`**`()`

Renvoie, sous forme de chaîne, la version de `pip` qui serait installée par `ensurepip`.

`ensurepip.`**`bootstrap`**`(`*root=None*, *upgrade=False*, *user=False*, *altinstall=False*, *default_pip=False*, *verbosity=0*`)`

Amorce `pip` dans l'environnement courant ou un environnement spécifique.

Passer *root* permet de changer la racine du chemin d'installation. Si *root* vaut `None` (la valeur par défaut), l'installation se fait dans la racine par défaut pour l'environnement courant.

upgrade indique s'il faut ou non effectuer la mise à jour d'une éventuelle version plus ancienne de `pip` déjà installée vers la version embarquée dans `ensurepip`.

Si *user* vaut vrai, `pip` est mis dans des chemins qui le rendent disponible pour cet utilisateur uniquement, et non pour tous les utilisateurs de l'installation de Python.

Par défaut, les commandes créées sont `pipX` et `pipX.Y` (où X.Y est la version de Python).

Si *altinstall* vaut vrai, `pipX` n'est pas créée.

Si *default_pip* vaut vrai, la commande `pip` est créée en plus des deux autres.

Le fait de combiner *altinstall* et *default_pip* lève l'exception *`ValueError`*.

verbosity règle le niveau de verbosité des messages émis sur *`sys.stdout`* pendant l'amorçage.

Cette fonction lève un *événement d'audit* `ensurepip.bootstrap` avec l'argument `root`.

> **Note**
>
> Le processus d'amorçage a des effets de bord sur `sys.path` et `os.environ`. Pour les éviter, on peut appeler l'interface en ligne de commande dans un sous-processus.

> **Note**
>
> L'amorçage peut installer des modules supplémentaires qui sont requis pour `pip`. Les autres programmes ne doivent pas prendre pour acquise la présence de ces modules, car `pip` pourrait dans une version future se passer de ces dépendances.

29.2 `venv` --- Creation of virtual environments

Ajouté dans la version 3.3.

Code source : Lib/venv/

The `venv` module supports creating lightweight "virtual environments", each with their own independent set of Python packages installed in their `site` directories. A virtual environment is created on top of an existing Python installation, known as the virtual environment's "base" Python, and may optionally be isolated from the packages in the base environment, so only those explicitly installed in the virtual environment are available. See *Virtual Environments* and `site`'s *virtual environments documentation* for more information.

When used from within a virtual environment, common installation tools such as pip will install Python packages into a virtual environment without needing to be told to do so explicitly.

A virtual environment is (amongst other things) :

- Used to contain a specific Python interpreter and software libraries and binaries which are needed to support a project (library or application). These are by default isolated from software in other virtual environments and Python interpreters and libraries installed in the operating system.
- Contained in a directory, conventionally named `.venv` or `venv` in the project directory, or under a container directory for lots of virtual environments, such as `~/.virtualenvs`.
- Not checked into source control systems such as Git.
- Considered as disposable -- it should be simple to delete and recreate it from scratch. You don't place any project code in the environment.
- Not considered as movable or copyable -- you just recreate the same environment in the target location.

See **PEP 405** for more background on Python virtual environments.

> **Voir aussi**
>
> Python Packaging User Guide : Creating and using virtual environments

Availability : not Android, not iOS, not WASI.

This module is not supported on *mobile platforms* or *WebAssembly platforms*.

29.2.1 Création d'environnements virtuels

Virtual environments are created by executing the `venv` module :

```
python -m venv /path/to/new/virtual/environment
```

This creates the target directory (including parent directories as needed) and places a `pyvenv.cfg` file in it with a `home` key pointing to the Python installation from which the command was run. It also creates a `bin` (or `Scripts` on Windows) subdirectory containing a copy or symlink of the Python executable (as appropriate for the platform or arguments used at environment creation time). It also creates a `lib/pythonX.Y/site-packages` subdirectory (on Windows, this is `Libsite-packages`). If an existing directory is specified, it will be re-used.

Modifié dans la version 3.5 : L'utilisation de `venv` est maintenant recommandée pour créer vos environnements virtuels.

Deprecated since version 3.6, removed in version 3.8 : **`pyvenv`** was the recommended tool for creating virtual environments for Python 3.3 and 3.4, and replaced in 3.5 by executing `venv` directly.

On Windows, invoke the `venv` command as follows :

```
PS> python -m venv C:\path\to\new\virtual\environment
```

La commande, si lancée avec `-h`, montrera les options disponibles :

```
usage: venv [-h] [--system-site-packages] [--symlinks | --copies] [--clear]
            [--upgrade] [--without-pip] [--prompt PROMPT] [--upgrade-deps]
            [--without-scm-ignore-files]
            ENV_DIR [ENV_DIR ...]

Creates virtual Python environments in one or more target directories.

positional arguments:
  ENV_DIR               A directory to create the environment in.

options:
  -h, --help            show this help message and exit
  --system-site-packages
                        Give the virtual environment access to the system
                        site-packages dir.
  --symlinks            Try to use symlinks rather than copies, when
                        symlinks are not the default for the platform.
  --copies              Try to use copies rather than symlinks, even when
                        symlinks are the default for the platform.
  --clear               Delete the contents of the environment directory
                        if it already exists, before environment creation.
  --upgrade             Upgrade the environment directory to use this
                        version of Python, assuming Python has been
                        upgraded in-place.
  --without-pip         Skips installing or upgrading pip in the virtual
                        environment (pip is bootstrapped by default)
  --prompt PROMPT       Provides an alternative prompt prefix for this
                        environment.
  --upgrade-deps        Upgrade core dependencies (pip) to the latest
                        version in PyPI
  --without-scm-ignore-files
                        Skips adding SCM ignore files to the environment
                        directory (Git is supported by default).

Once an environment has been created, you may wish to activate it, e.g. by
sourcing an activate script in its bin directory.
```

Modifié dans la version 3.4 : Installs pip by default, added the `--without-pip` and `--copies` options.

Modifié dans la version 3.4 : Dans les versions précédentes, si le dossier de destination existait déjà, une erreur était levée, sauf si l'option `--clear` ou `--upgrade` était incluse.

Modifié dans la version 3.9 : Add `--upgrade-deps` option to upgrade pip + setuptools to the latest on PyPI.

Modifié dans la version 3.12 : `setuptools` is no longer a core venv dependency.

Modifié dans la version 3.13 : Added the `--without-scm-ignore-files` option.

Modifié dans la version 3.13 : `venv` now creates a `.gitignore` file for Git by default.

Note

Bien que les liens symboliques soient pris en charge sous Windows, ils ne sont pas recommandés. Il est particulièrement à noter que le double-clic sur `python.exe` dans l'Explorateur de fichiers suivra le lien symbolique et

ignorera l'environnement virtuel.

Note

On Microsoft Windows, it may be required to enable the `Activate.ps1` script by setting the execution policy for the user. You can do this by issuing the following PowerShell command :

```
PS C:\> Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope CurrentUser
```

See About Execution Policies for more information.

The created `pyvenv.cfg` file also includes the `include-system-site-packages` key, set to `true` if `venv` is run with the `--system-site-packages` option, `false` otherwise.

Sauf si l'option `--without-pip` est incluse, *ensurepip* sera invoqué pour amorcer `pip` dans l'environnement virtuel.

Plusieurs chemins peuvent être donnés à `venv`, et dans ce cas un environnement virtuel sera créé, en fonction des options incluses, à chaque chemin donné.

29.2.2 How venvs work

When a Python interpreter is running from a virtual environment, *sys.prefix* and *sys.exec_prefix* point to the directories of the virtual environment, whereas *sys.base_prefix* and *sys.base_exec_prefix* point to those of the base Python used to create the environment. It is sufficient to check `sys.prefix != sys.base_prefix` to determine if the current interpreter is running from a virtual environment.

A virtual environment may be "activated" using a script in its binary directory (`bin` on POSIX ; `Scripts` on Windows). This will prepend that directory to your `PATH`, so that running **python** will invoke the environment's Python interpreter and you can run installed scripts without having to use their full path. The invocation of the activation script is platform-specific (*<venv>* must be replaced by the path to the directory containing the virtual environment) :

Plateforme	Invite de commande	Commande pour activer l'environnement virtuel
POSIX	bash/zsh	`$ source <venv>/bin/activate`
	fish	`$ source <venv>/bin/activate.fish`
	csh/tcsh	`$ source <venv>/bin/activate.csh`
	pwsh	`$ <venv>/bin/Activate.ps1`
Windows	cmd.exe	`C:\> <venv>\Scripts\activate.bat`
	PowerShell	`PS C:\> <venv>\Scripts\Activate.ps1`

Ajouté dans la version 3.4 : **fish** and **csh** activation scripts.

Ajouté dans la version 3.8 : Scripts d'activation PowerShell installés sous POSIX pour le support de PowerShell Core.

You don't specifically *need* to activate a virtual environment, as you can just specify the full path to that environment's Python interpreter when invoking Python. Furthermore, all scripts installed in the environment should be runnable without activating it.

In order to achieve this, scripts installed into virtual environments have a "shebang" line which points to the environment's Python interpreter, `#!/<path-to-venv>/bin/python`. This means that the script will run with that interpreter regardless of the value of `PATH`. On Windows, "shebang" line processing is supported if you have the launcher installed. Thus, double-clicking an installed script in a Windows Explorer window should run it with the correct interpreter without the environment needing to be activated or on the `PATH`.

When a virtual environment has been activated, the `VIRTUAL_ENV` environment variable is set to the path of the environment. Since explicitly activating a virtual environment is not required to use it, `VIRTUAL_ENV` cannot be relied upon to determine whether a virtual environment is being used.

Avertissement

Because scripts installed in environments should not expect the environment to be activated, their shebang lines contain the absolute paths to their environment's interpreters. Because of this, environments are inherently non-portable, in the general case. You should always have a simple means of recreating an environment (for example, if you have a requirements file `requirements.txt`, you can invoke `pip install -r requirements.txt` using the environment's `pip` to install all of the packages needed by the environment). If for any reason you need to move the environment to a new location, you should recreate it at the desired location and delete the one at the old location. If you move an environment because you moved a parent directory of it, you should recreate the environment in its new location. Otherwise, software installed into the environment may not work as expected.

You can deactivate a virtual environment by typing `deactivate` in your shell. The exact mechanism is platform-specific and is an internal implementation detail (typically, a script or shell function will be used).

29.2.3 API

La méthode haut niveau décrite au dessus utilise une API simple qui permet à des créateurs d'environnements virtuels externes de personnaliser la création d'environnements virtuels basés sur leurs besoins, la classe `EnvBuilder`.

class venv.**EnvBuilder**(*system_site_packages=False*, *clear=False*, *symlinks=False*, *upgrade=False*, *with_pip=False*, *prompt=None*, *upgrade_deps=False*, **, scm_ignore_files=frozenset()*)

La classe `EnvBuilder` accepte les arguments suivants lors de l'instanciation :

- *system_site_packages* -- a boolean value indicating that the system Python site-packages should be available to the environment (defaults to `False`).
- *clear* -- a boolean value which, if true, will delete the contents of any existing target directory, before creating the environment.
- *symlinks* -- a boolean value indicating whether to attempt to symlink the Python binary rather than copying.
- *upgrade* -- a boolean value which, if true, will upgrade an existing environment with the running Python - for use when that Python has been upgraded in-place (defaults to `False`).
- *with_pip* -- a boolean value which, if true, ensures pip is installed in the virtual environment. This uses `ensurepip` with the `--default-pip` option.
- *prompt* -- a string to be used after virtual environment is activated (defaults to `None` which means directory name of the environment would be used). If the special string `"."` is provided, the basename of the current directory is used as the prompt.
- *upgrade_deps* -- Update the base venv modules to the latest on PyPI
- *scm_ignore_files* -- Create ignore files based for the specified source control managers (SCM) in the iterable. Support is defined by having a method named `create_{scm}_ignore_file`. The only value supported by default is `"git"` via `create_git_ignore_file()`.

Modifié dans la version 3.4 : Ajout du paramètre `with_pip`

Modifié dans la version 3.6 : Ajout du paramètre `prompt`

Modifié dans la version 3.9 : Added the `upgrade_deps` parameter

Modifié dans la version 3.13 : Added the `scm_ignore_files` parameter

`EnvBuilder` may be used as a base class.

create(*env_dir*)

Crée un environnement virtuel en spécifiant le chemin cible (absolu ou relatif par rapport au dossier courant) qui contiendra l'environnement virtuel. La méthode `create` crée l'environnement dans le dossier spécifié ou lève une exception appropriée.

La méthode `create` de la classe `EnvBuilder` illustre les points d'entrées disponibles pour la personnalisation de sous-classes :

```
def create(self, env_dir):
    """
    Create a virtualized Python environment in a directory.
    env_dir is the target directory to create an environment in.
```

(suite sur la page suivante)

(suite de la page précédente)

```
    """
    env_dir = os.path.abspath(env_dir)
    context = self.ensure_directories(env_dir)
    self.create_configuration(context)
    self.setup_python(context)
    self.setup_scripts(context)
    self.post_setup(context)
```

Chacune des méthodes *ensure_directories()*, *create_configuration()*, *setup_python()*, *setup_scripts()* et *post_setup()* peuvent être écrasés.

ensure_directories(*env_dir*)

Creates the environment directory and all necessary subdirectories that don't already exist, and returns a context object. This context object is just a holder for attributes (such as paths) for use by the other methods. If the *EnvBuilder* is created with the arg `clear=True`, contents of the environment directory will be cleared and then all necessary subdirectories will be recreated.

The returned context object is a *types.SimpleNamespace* with the following attributes :

- `env_dir` - The location of the virtual environment. Used for `__VENV_DIR__` in activation scripts (see *install_scripts()*).
- `env_name` - The name of the virtual environment. Used for `__VENV_NAME__` in activation scripts (see *install_scripts()*).
- `prompt` - The prompt to be used by the activation scripts. Used for `__VENV_PROMPT__` in activation scripts (see *install_scripts()*).
- `executable` - The underlying Python executable used by the virtual environment. This takes into account the case where a virtual environment is created from another virtual environment.
- `inc_path` - The include path for the virtual environment.
- `lib_path` - The purelib path for the virtual environment.
- `bin_path` - The script path for the virtual environment.
- `bin_name` - The name of the script path relative to the virtual environment location. Used for `__VENV_BIN_NAME__` in activation scripts (see *install_scripts()*).
- `env_exe` - The name of the Python interpreter in the virtual environment. Used for `__VENV_PYTHON__` in activation scripts (see *install_scripts()*).
- `env_exec_cmd` - The name of the Python interpreter, taking into account filesystem redirections. This can be used to run Python in the virtual environment.

Modifié dans la version 3.11 : The *venv sysconfig installation scheme* is used to construct the paths of the created directories.

Modifié dans la version 3.12 : The attribute `lib_path` was added to the context, and the context object was documented.

create_configuration(*context*)

Crée le fichier de configuration `pyvenv.cfg` dans l'environnement.

setup_python(*context*)

Crée une copie ou un lien symbolique vers l'exécutable Python dans l'environnement. Sur les systèmes POSIX, si un exécutable spécifique `python3.x` a été utilisé, des liens symboliques vers `python` et `python3` seront créés pointant vers cet exécutable, sauf si des fichiers avec ces noms existent déjà.

setup_scripts(*context*)

Installe les scripts d'activation appropriés à la plateforme dans l'environnement virtuel.

upgrade_dependencies(*context*)

Upgrades the core venv dependency packages (currently pip) in the environment. This is done by shelling out to the `pip` executable in the environment.

Ajouté dans la version 3.9.

Modifié dans la version 3.12 : setuptools is no longer a core venv dependency.

post_setup(*context*)

Une méthode qui n'est la que pour se faire surcharger dans des implémentation externes pour pré installer des paquets dans l'environnement virtuel ou pour exécuter des étapes post-création.

install_scripts(*context*, *path*)

This method can be called from `setup_scripts()` or `post_setup()` in subclasses to assist in installing custom scripts into the virtual environment.

path is the path to a directory that should contain subdirectories `common`, `posix`, `nt` ; each containing scripts destined for the `bin` directory in the environment. The contents of `common` and the directory corresponding to `os.name` are copied after some text replacement of placeholders :

— `__VENV_DIR__` est remplacé avec le chemin absolu du dossier de l'environnement.
— `__VENV_NAME__` est remplacé avec le nom de l'environnement (le dernier segment du chemin vers le dossier de l'environnement).
— `__VENV_PROMPT__` est remplacé par le prompt (nom de l'environnement entouré de parenthèses et avec un espace le suivant).
— `__VENV_BIN_NAME__` est remplacé par le nom du dossier **bin** (soit `bin` soit `Scripts`).
— `__VENV_PYTHON__` est remplacé avec le chemin absolu de l'exécutable de l'environnement.

Les dossiers peuvent exister (pour quand un environnement existant est mis à jour).

create_git_ignore_file(*context*)

Creates a `.gitignore` file within the virtual environment that causes the entire directory to be ignored by the Git source control manager.

Ajouté dans la version 3.13.

Modifié dans la version 3.7.2 : Windows utilise maintenant des scripts de redirection pour `python[w].exe` au lieu de copier les fichiers binaires. En 3.7.2 seulement `setup_python()` ne fait rien sauf s'il s'exécute à partir d'un *build* dans l'arborescence source.

Modifié dans la version 3.7.3 : Windows copie les scripts de redirection dans le cadre de `setup_python()` au lieu de `setup_scripts()`. Ce n'était pas le cas en 3.7.2. Lorsque vous utilisez des liens symboliques, les exécutables originaux seront liés.

Il y a aussi une fonction pratique au niveau du module :

venv.**create**(*env_dir*, *system_site_packages=False*, *clear=False*, *symlinks=False*, *with_pip=False*, *prompt=None*, *upgrade_deps=False*, *, *scm_ignore_files=frozenset()*)

Crée une `EnvBuilder` avec les arguments donnés, et appelle sa méthode `create()` avec l'argument *env_dir*.

Ajouté dans la version 3.3.

Modifié dans la version 3.4 : Added the *with_pip* parameter

Modifié dans la version 3.6 : Added the *prompt* parameter

Modifié dans la version 3.9 : Added the *upgrade_deps* parameter

Modifié dans la version 3.13 : Added the *scm_ignore_files* parameter

29.2.4 Un exemple d'extension de `EnvBuilder`

Le script qui suis montre comment étendre `EnvBuilder` en implémentant une sous-classe qui installe **setuptools** et **pip** dans un environnement créé :

```
import os
import os.path
from subprocess import Popen, PIPE
import sys
from threading import Thread
from urllib.parse import urlparse
from urllib.request import urlretrieve
import venv
```

(suite sur la page suivante)

(suite de la page précédente)

```
class ExtendedEnvBuilder(venv.EnvBuilder):
    """
    This builder installs setuptools and pip so that you can pip or
    easy_install other packages into the created virtual environment.

    :param nodist: If true, setuptools and pip are not installed into the
                   created virtual environment.
    :param nopip: If true, pip is not installed into the created
                  virtual environment.
    :param progress: If setuptools or pip are installed, the progress of the
                     installation can be monitored by passing a progress
                     callable. If specified, it is called with two
                     arguments: a string indicating some progress, and a
                     context indicating where the string is coming from.
                     The context argument can have one of three values:
                     'main', indicating that it is called from virtualize()
                     itself, and 'stdout' and 'stderr', which are obtained
                     by reading lines from the output streams of a subprocess
                     which is used to install the app.

                     If a callable is not specified, default progress
                     information is output to sys.stderr.
    """

    def __init__(self, *args, **kwargs):
        self.nodist = kwargs.pop('nodist', False)
        self.nopip = kwargs.pop('nopip', False)
        self.progress = kwargs.pop('progress', None)
        self.verbose = kwargs.pop('verbose', False)
        super().__init__(*args, **kwargs)

    def post_setup(self, context):
        """
        Set up any packages which need to be pre-installed into the
        virtual environment being created.

        :param context: The information for the virtual environment
                        creation request being processed.
        """
        os.environ['VIRTUAL_ENV'] = context.env_dir
        if not self.nodist:
            self.install_setuptools(context)
        # Can't install pip without setuptools
        if not self.nopip and not self.nodist:
            self.install_pip(context)

    def reader(self, stream, context):
        """
        Read lines from a subprocess' output stream and either pass to a progress
        callable (if specified) or write progress information to sys.stderr.
        """
        progress = self.progress
        while True:
            s = stream.readline()
            if not s:
                break
```

(suite sur la page suivante)

(suite de la page précédente)

```
            if progress is not None:
                progress(s, context)
            else:
                if not self.verbose:
                    sys.stderr.write('.')
                else:
                    sys.stderr.write(s.decode('utf-8'))
                sys.stderr.flush()
        stream.close()

    def install_script(self, context, name, url):
        _, _, path, _, _, _ = urlparse(url)
        fn = os.path.split(path)[-1]
        binpath = context.bin_path
        distpath = os.path.join(binpath, fn)
        # Download script into the virtual environment's binaries folder
        urlretrieve(url, distpath)
        progress = self.progress
        if self.verbose:
            term = '\n'
        else:
            term = ''
        if progress is not None:
            progress('Installing %s ...%s' % (name, term), 'main')
        else:
            sys.stderr.write('Installing %s ...%s' % (name, term))
            sys.stderr.flush()
        # Install in the virtual environment
        args = [context.env_exe, fn]
        p = Popen(args, stdout=PIPE, stderr=PIPE, cwd=binpath)
        t1 = Thread(target=self.reader, args=(p.stdout, 'stdout'))
        t1.start()
        t2 = Thread(target=self.reader, args=(p.stderr, 'stderr'))
        t2.start()
        p.wait()
        t1.join()
        t2.join()
        if progress is not None:
            progress('done.', 'main')
        else:
            sys.stderr.write('done.\n')
        # Clean up - no longer needed
        os.unlink(distpath)

    def install_setuptools(self, context):
        """
        Install setuptools in the virtual environment.

        :param context: The information for the virtual environment
                        creation request being processed.
        """
        url = "https://bootstrap.pypa.io/ez_setup.py"
        self.install_script(context, 'setuptools', url)
        # clear up the setuptools archive which gets downloaded
        pred = lambda o: o.startswith('setuptools-') and o.endswith('.tar.gz')
        files = filter(pred, os.listdir(context.bin_path))
```

(suite sur la page suivante)

(suite de la page précédente)

```
        for f in files:
            f = os.path.join(context.bin_path, f)
            os.unlink(f)

    def install_pip(self, context):
        """
        Install pip in the virtual environment.

        :param context: The information for the virtual environment
                        creation request being processed.
        """
        url = 'https://bootstrap.pypa.io/get-pip.py'
        self.install_script(context, 'pip', url)


def main(args=None):
    import argparse

    parser = argparse.ArgumentParser(prog=__name__,
                                     description='Creates virtual Python '
                                                 'environments in one or '
                                                 'more target '
                                                 'directories.')
    parser.add_argument('dirs', metavar='ENV_DIR', nargs='+',
                        help='A directory in which to create the '
                             'virtual environment.')
    parser.add_argument('--no-setuptools', default=False,
                        action='store_true', dest='nodist',
                        help="Don't install setuptools or pip in the "
                             "virtual environment.")
    parser.add_argument('--no-pip', default=False,
                        action='store_true', dest='nopip',
                        help="Don't install pip in the virtual "
                             "environment.")
    parser.add_argument('--system-site-packages', default=False,
                        action='store_true', dest='system_site',
                        help='Give the virtual environment access to the '
                             'system site-packages dir.')
    if os.name == 'nt':
        use_symlinks = False
    else:
        use_symlinks = True
    parser.add_argument('--symlinks', default=use_symlinks,
                        action='store_true', dest='symlinks',
                        help='Try to use symlinks rather than copies, '
                             'when symlinks are not the default for '
                             'the platform.')
    parser.add_argument('--clear', default=False, action='store_true',
                        dest='clear', help='Delete the contents of the '
                                           'virtual environment '
                                           'directory if it already '
                                           'exists, before virtual '
                                           'environment creation.')
    parser.add_argument('--upgrade', default=False, action='store_true',
                        dest='upgrade', help='Upgrade the virtual '
                                             'environment directory to '
```

(suite sur la page suivante)

(suite de la page précédente)

```
                                                 'use this version of '
                                                 'Python, assuming Python '
                                                 'has been upgraded '
                                                 'in-place.')
    parser.add_argument('--verbose', default=False, action='store_true',
                        dest='verbose', help='Display the output '
                                                 'from the scripts which '
                                                 'install setuptools and pip.')
    options = parser.parse_args(args)
    if options.upgrade and options.clear:
        raise ValueError('you cannot supply --upgrade and --clear together.')
    builder = ExtendedEnvBuilder(system_site_packages=options.system_site,
                                 clear=options.clear,
                                 symlinks=options.symlinks,
                                 upgrade=options.upgrade,
                                 nodist=options.nodist,
                                 nopip=options.nopip,
                                 verbose=options.verbose)
    for d in options.dirs:
        builder.create(d)

if __name__ == '__main__':
    rc = 1
    try:
        main()
        rc = 0
    except Exception as e:
        print('Error: %s' % e, file=sys.stderr)
    sys.exit(rc)
```

Ce script est aussi disponible au téléchargement en ligne.

29.3 `zipapp` --- Manage executable Python zip archives

Ajouté dans la version 3.5.

Code source : Lib/zipapp.py

Ce module fournit des outils pour gérer la création de fichiers zip contenant du code Python, qui peuvent être exécutés directement par l'interpréteur Python. Le module fournit à la fois une interface de ligne de commande *Interface en ligne de commande* et une interface *API Python*.

29.3.1 Exemple de base

L'exemple suivant montre comment l'interface de ligne de commande *Interface en ligne de commande* peut être utilisée pour créer une archive exécutable depuis un répertoire contenant du code Python. Lors de l'exécution, l'archive exécutera la fonction `main` du module `myapp` dans l'archive.

```
$ python -m zipapp myapp -m "myapp:main"
$ python myapp.pyz
<output from myapp>
```

29.3.2 Interface en ligne de commande

Lorsqu'il est appelé en tant que programme à partir de la ligne de commande, la syntaxe suivante est utilisée :

```
$ python -m zipapp source [options]
```

Si *source* est un répertoire, une archive est créée à partir du contenu de *source*. Si *source* est un fichier, ce doit être une archive et il est copié dans l'archive cible (ou le contenu de sa ligne *shebang* est affiché si l'option `--info` est indiquée).

Les options suivantes sont disponibles :

-o `<output>`, **--output**=`<output>`

Écrit la sortie dans un fichier nommé *output*. Si cette option n'est pas spécifiée, le nom du fichier de sortie sera le même que celui de l'entrée *source*, avec l'extension `.pyz`. Si un nom de fichier explicite est donné, il est utilisé tel quel (une extension `.pyz` doit donc être incluse si nécessaire).

Un nom de fichier de sortie doit être spécifié si la *source* est une archive (et, dans ce cas, la *sortie* ne doit pas être la même que la *source*).

-p `<interpreter>`, **--python**=`<interpreter>`

Ajoute une ligne `#!` à l'archive en spécifiant *interpreter* comme commande à exécuter. Aussi, sur un système POSIX, cela rend l'archive exécutable. Le comportement par défaut est de ne pas écrire la ligne `#!` et de ne pas rendre le fichier exécutable.

-m `<mainfn>`, **--main**=`<mainfn>`

Écrit un fichier `__main__.py` dans l'archive qui exécute *mainfn*. L'argument *mainfn* est de la forme « *pkg.mod :fn* », où « *pkg.mod* » est un paquet/module dans l'archive, et « *fn* » est un appelable dans le module donné. Le fichier `__main__.py` réalise cet appel.

`--main` ne peut pas être spécifié lors de la copie d'une archive.

-c, --compress

Compresse les fichiers avec la méthode *deflate*, réduisant ainsi la taille du fichier de sortie. Par défaut, les fichiers sont stockés non compressés dans l'archive.

`--compress` n'a aucun effet lors de la copie d'une archive.

Ajouté dans la version 3.7.

--info

Affiche l'interpréteur intégré dans l'archive, à des fins de diagnostic. Dans ce cas, toutes les autres options sont ignorées et SOURCE doit être une archive et non un répertoire.

-h, --help

Affiche un court message d'aide et quitte.

29.3.3 API Python

Ce module définit deux fonctions utilitaires :

`zipapp.`**`create_archive`**(*source*, *target=None*, *interpreter=None*, *main=None*, *filter=None*, *compressed=False*)

Crée une archive d'application à partir de *source*. La source peut être de natures suivantes :

— Le nom d'un répertoire, ou un *path-like object* se référant à un répertoire ; dans ce cas, une nouvelle archive d'application sera créée à partir du contenu de ce répertoire.

— Le nom d'un fichier d'archive d'application existant, ou un *path-like object* se référant à un tel fichier ; dans ce cas, le fichier est copié sur la cible (en le modifiant pour refléter la valeur donnée à l'argument *interpreter*). Le nom du fichier doit inclure l'extension `.pyz`, si nécessaire.

— Un objet fichier ouvert pour la lecture en mode binaire. Le contenu du fichier doit être une archive d'application et Python suppose que l'objet fichier est positionné au début de l'archive.

L'argument *target* détermine où l'archive résultante sera écrite :

— S'il s'agit d'un nom de fichier, ou d'un *path-like object*, l'archive sera écrite dans ce fichier.

— S'il s'agit d'un objet fichier ouvert, l'archive sera écrite dans cet objet fichier, qui doit être ouvert pour l'écriture en mode octets.
— Si la cible est omise (ou `None`), la source doit être un répertoire et la cible sera un fichier portant le même nom que la source, avec une extension `.pyz` ajoutée.

L'argument *interpreter* spécifie le nom de l'interpréteur Python avec lequel l'archive sera exécutée. Il est écrit dans une ligne *shebang* au début de l'archive. Sur un système POSIX, cela est interprété par le système d'exploitation et, sur Windows, il sera géré par le lanceur Python. L'omission de l'*interpreter* n'entraîne pas l'écriture d'une ligne *shebang*. Si un interpréteur est spécifié et que la cible est un nom de fichier, le bit exécutable du fichier cible sera mis à 1.

L'argument *main* spécifie le nom d'un appelable, utilisé comme programme principal pour l'archive. Il ne peut être spécifié que si la source est un répertoire et si la source ne contient pas déjà un fichier `__main__.py`. L'argument *main* doit prendre la forme `pkg.module:callable` et l'archive sera exécutée en important `pkg.module` et en exécutant l'appelable donné sans argument. Omettre *main* est une erreur si la source est un répertoire et ne contient pas un fichier `__main__.py` car, dans ce cas, l'archive résultante ne serait pas exécutable.

L'argument optionnel *filter* spécifie une fonction de rappel à laquelle on passe un objet *Path* représentant le chemin du fichier à ajouter (par rapport au répertoire source). Elle doit renvoyer `True` si le fichier doit effectivement être ajouté.

L'argument optionnel *compressed* détermine si les fichiers doivent être compressés. S'il vaut `True`, les fichiers de l'archive sont compressés avec l'algorithme *deflate* ; sinon, les fichiers sont stockés non compressés. Cet argument n'a aucun effet lors de la copie d'une archive existante.

Si un objet fichier est spécifié pour *source* ou *target*, il est de la responsabilité de l'appelant de le fermer après avoir appelé `create_archive`.

Lors de la copie d'une archive existante, les objets fichier fournis n'ont besoin que des méthodes `read` et `readline` ou `write`. Lors de la création d'une archive à partir d'un répertoire, si la cible est un objet fichier, elle sera passée à la classe `zipfile.ZipFile` et devra fournir les méthodes nécessaires à cette classe.

Modifié dans la version 3.7 : Added the *filter* and *compressed* parameters.

`zipapp.`**`get_interpreter`**(*archive*)

Renvoie l'interpréteur spécifié dans la ligne `#!` au début de l'archive. S'il n'y a pas de ligne `#!`, renvoie `None`. L'argument *archive* peut être un nom de fichier ou un objet de type fichier ouvert à la lecture en mode binaire. Python suppose qu'il est au début de l'archive.

29.3.4 Exemples

Regroupe le contenu d'un répertoire dans une archive, puis l'exécute.

```
$ python -m zipapp myapp
$ python myapp.pyz
<output from myapp>
```

La même chose peut être faite en utilisant la fonction `create_archive()` :

```
>>> import zipapp
>>> zipapp.create_archive('myapp', 'myapp.pyz')
```

Pour rendre l'application directement exécutable sur un système POSIX, spécifiez un interpréteur à utiliser.

```
$ python -m zipapp myapp -p "/usr/bin/env python"
$ ./myapp.pyz
<output from myapp>
```

Pour remplacer la ligne *shebang* sur une archive existante, créez une archive modifiée en utilisant la fonction `create_archive()` :

```
>>> import zipapp
>>> zipapp.create_archive('old_archive.pyz', 'new_archive.pyz', '/usr/bin/python3')
```

To update the file in place, do the replacement in memory using a `BytesIO` object, and then overwrite the source afterwards. Note that there is a risk when overwriting a file in place that an error will result in the loss of the original file. This code does not protect against such errors, but production code should do so. Also, this method will only work if the archive fits in memory :

```
>>> import zipapp
>>> import io
>>> temp = io.BytesIO()
>>> zipapp.create_archive('myapp.pyz', temp, '/usr/bin/python2')
>>> with open('myapp.pyz', 'wb') as f:
>>>     f.write(temp.getvalue())
```

29.3.5 Spécification de l'interprète

Notez que si vous spécifiez un interpréteur et que vous distribuez ensuite votre archive d'application, vous devez vous assurer que l'interpréteur utilisé est portable. Le lanceur Python pour Windows gère la plupart des formes courantes de la ligne POSIX `#!`, mais il y a d'autres problèmes à considérer :

- Si vous utilisez `/usr/bin/env python` (ou d'autres formes de la commande *python*, comme `/usr/bin/python`), vous devez considérer que vos utilisateurs peuvent avoir Python 2 ou Python 3 par défaut, et écrire votre code pour fonctionner dans les deux versions.
- Si vous utilisez une version explicite, par exemple `/usr/bin/env python3` votre application ne fonctionnera pas pour les utilisateurs qui n'ont pas cette version. (C'est peut-être ce que vous voulez si vous n'avez pas rendu votre code compatible Python 2).
- Il n'y a aucun moyen de dire « python X.Y ou supérieur » donc faites attention si vous utilisez une version exacte comme `/usr/bin/env python3.4` car vous devrez changer votre ligne *shebang* pour les utilisateurs de Python 3.5, par exemple.

Normalement, vous devriez utiliser un `/usr/bin/env python2` ou `/usr/bin/env python3`, selon que votre code soit écrit pour Python 2 ou 3.

29.3.6 Création d'applications autonomes avec *zipapp*

En utilisant le module `zipapp`, il est possible de créer des programmes Python qui peuvent être distribués à des utilisateurs finaux dont le seul pré-requis est d'avoir la bonne version de Python installée sur leur ordinateur. Pour y arriver, la clé est de regrouper toutes les dépendances de l'application dans l'archive avec le code source de l'application.

Les étapes pour créer une archive autonome sont les suivantes :

1. Créez votre application dans un répertoire comme d'habitude, de manière à avoir un répertoire `myapp` contenant un fichier `__main__.py` et tout le code de l'application correspondante.
2. Installez toutes les dépendances de votre application dans le répertoire `myapp` en utilisant *pip* :

   ```
   $ python -m pip install -r requirements.txt --target myapp
   ```

   (ceci suppose que vous ayez vos dépendances de projet dans un fichier `requirements.txt` — sinon vous pouvez simplement lister les dépendances manuellement sur la ligne de commande *pip*).
3. Regroupez le tout à l'aide de :

   ```
   $ python -m zipapp -p "interpreter" myapp
   ```

Cela produira un exécutable autonome qui peut être exécuté sur n'importe quelle machine avec l'interpréteur approprié disponible. Voir *Spécification de l'interprète* pour plus de détails. Il peut être envoyé aux utilisateurs sous la forme d'un seul fichier.

Sous Unix, le fichier `myapp.pyz` est exécutable tel quel. Vous pouvez renommer le fichier pour supprimer l'extension `.pyz` si vous préférez un nom de commande « simple ». Sous Windows, le fichier `myapp.pyz[w]` est exécutable en vertu du fait que l'interpréteur Python est associé aux extensions de fichier `.pyz` et `.pyzw` une fois installé.

Mises en garde

Si votre application dépend d'un paquet qui inclut une extension C, ce paquet ne peut pas être exécuté à partir d'un fichier zip (c'est une limitation du système d'exploitation, car le code exécutable doit être présent dans le système de fichiers pour que le lanceur de l'OS puisse le charger). Dans ce cas, vous pouvez exclure cette dépendance du fichier zip et, soit demander à vos utilisateurs de l'installer, soit la fournir avec votre fichier zip et ajouter du code à votre fichier `__main__.py` pour inclure le répertoire contenant le module décompressé dans `sys.path`. Dans ce cas, vous devrez vous assurer d'envoyer les binaires appropriés pour votre ou vos architecture(s) cible(s) (et éventuellement choisir la bonne version à ajouter à `sys.path` au moment de l'exécution, basée sur la machine de l'utilisateur).

29.3.7 Le format d'archive d'application Zip Python

Python est capable d'exécuter des fichiers zip qui contiennent un fichier `__main__.py` depuis la version 2.6. Pour être exécutée par Python, une archive d'application doit simplement être un fichier zip standard contenant un fichier `__main__.py` qui sera exécuté comme point d'entrée de l'application. Comme d'habitude pour tout script Python, le parent du script (dans ce cas le fichier zip) sera placé sur *sys.path* et ainsi d'autres modules pourront être importés depuis le fichier zip.

Le format de fichier zip permet d'ajouter des données arbitraires à un fichier zip. Le format de l'application zip utilise cette possibilité pour préfixer une ligne *shebang* POSIX standard dans le fichier (`#!/path/to/interpreter`).

Formellement, le format d'application zip de Python est donc :

1. An optional shebang line, containing the characters `b'#!'` followed by an interpreter name, and then a newline (`b'\n'`) character. The interpreter name can be anything acceptable to the OS "shebang" processing, or the Python launcher on Windows. The interpreter should be encoded in UTF-8 on Windows, and in *sys.getfilesystemencoding()* on POSIX.
2. Des données *zipfile* standards, telles que générées par le module *zipfile*. Le contenu du fichier zip *doit* inclure un fichier appelé `__main__.py` (qui doit se trouver à la racine du fichier zip — c'est-à-dire qu'il ne peut se trouver dans un sous-répertoire). Les données du fichier zip peuvent être compressées ou non.

Si une archive d'application a une ligne *shebang*, elle peut avoir le bit exécutable activé sur les systèmes POSIX, pour lui permettre d'être exécutée directement.

Vous pouvez créer des archives d'applications sans utiliser les outils de ce module — le module existe pour faciliter les choses, mais les archives, créées par n'importe quel moyen tout en respectant le format ci-dessus, sont valides pour Python.

CHAPITRE 30

Environnement d'exécution Python

Les modules décrits dans ce chapitre fournissent une large collection de services relatifs à l'interpréteur Python et son interaction avec son environnement. En voici un survol :

30.1 `sys` --- System-specific parameters and functions

This module provides access to some variables used or maintained by the interpreter and to functions that interact strongly with the interpreter. It is always available. Unless explicitly noted otherwise, all variables are read-only.

`sys.`**`abiflags`**

Contient, sur les systèmes POSIX où Python a été compilé avec le script `configure`, les *ABI flags* tels que définis par la **PEP 3149**.

Ajouté dans la version 3.2.

Modifié dans la version 3.8 : Les options par défaut sont devenues des chaines de caractères vides (l'option `m` pour *pymalloc* a été enlevée).

Availability : Unix.

`sys.`**`addaudithook`**`(`*hook*`)`

Append the callable *hook* to the list of active auditing hooks for the current (sub)interpreter.

Quand un événement d'audit est déclenché par la fonction `sys.audit()`, chaque point d'entrée est appelé dans l'ordre dans lequel il a été ajouté avec le nom de l'événement et le *n*-uplet des arguments. Les points d'entrées qui sont ajoutés par `PySys_AddAuditHook()` sont appelés les premiers, suivi par les fonctions de rappel ajoutées dans l'interpréteur en cours d'exécution. Les points d'entrées peuvent *logger* l'événement, lever une exception pour stopper l'opération ou terminer le processus entièrement.

Note that audit hooks are primarily for collecting information about internal or otherwise unobservable actions, whether by Python or libraries written in Python. They are not suitable for implementing a "sandbox". In particular, malicious code can trivially disable or bypass hooks added using this function. At a minimum, any security-sensitive hooks must be added using the C API `PySys_AddAuditHook()` before initialising the runtime, and any modules allowing arbitrary memory modification (such as `ctypes`) should be completely removed or closely monitored.

L'appel de `sys.addaudithook()` lèvera d'elle même un événement d'audit appelé `sys.addaudithook` sans arguments. Si n'importe quel *hook* lève une exception dérivée de `RuntimeError`, le nouveau point d'entrée ne

sera pas ajouté et l'exception supprimée. Par conséquent, les appels ne peuvent pas supposer que leurs points d'entrées ont été ajoutés à moins de contrôler tous les points d'entrées existants.

Voir la *table d'évenements d'audit* pour tous les événements levés par CPython et **PEP 578** pour la discussion originale de ce design.

Ajouté dans la version 3.8.

Modifié dans la version 3.8.1 : Les exceptions dérivées de `Exception` mais pas `RuntimeError` ne sont plus supprimées.

Quand le traçage est activé (voir `settrace()`), les *hooks* Python ne sont tracés que si la fonction a un membre `__cantrace__` dont la valeur est vraie. Autrement, les fonctions de traçage ignorent le *hook*.

`sys.`**`argv`**

La liste des arguments de la ligne de commande passés à un script Python. `argv[0]` est le nom du script (chemin complet, ou non, en fonction du système d'exploitation). Si la commande a été exécutée avec l'option `-c` de l'interpréteur, `argv[0]` vaut la chaîne `'-c'`. Si aucun nom de script n'a été donné à l'interpréteur Python, `argv[0]` sera une chaîne vide.

Pour boucler sur l'entrée standard, ou la liste des fichiers donnés sur la ligne de commande, utilisez le module `fileinput`.

See also `sys.orig_argv`.

> **Note**
>
> Sous Unix, les arguments de ligne de commande sont passés par des octets depuis le système d'exploitation. Python les décode en utilisant l'encodage du système de fichiers et le gestionnaire d'erreur *surrogateescape*. Quand vous avez besoin des octets originaux, vous pouvez les récupérer avec `[os.fsencode(arg) for arg in sys.argv]`.

`sys.`**`audit`**`(`*event*, **args*`)`

Déclenche un événement d'audit pour tous les points d'entrées d'audit actifs. *event* est une chaîne de caractères identifiant l'événement, et *args* peut contenir des arguments optionnels avec plus d'informations sur l'événement en question. Le nombre et types d'arguments pour un événement donné sont considérés comme une API publique et stable, et ne devraient pas être modifiés entre les versions.

Par exemple, un événement d'audit est nommé `os.chdir`. Cet événement à un argument appelé *path* qui contiendras le nouveau répertoire de travail.

`sys.audit()` appellera les point d'entrées (*hooks* en anglais) d'audits existants, en passant le nom de l'événement et ses arguments, et lèvera à nouveau la première exception de n'importe quel point d'entrée. En général, si une exception est levée, elle ne devrais pas être gérée et le processus devrait être terminé aussi rapidement que possible. Cela permet que les implémentations des points d'entrées décident comment répondre à des événements particuliers : Ils peuvent simplement *logger* l'événement ou arrêter l'opération en levant une exception.

Les points d'entrées sont ajoutés en utilisant les fonctions `sys.addaudithook()` ou `PySys_AddAuditHook()`.

L'équivalent natif de cette fonction est `PySys_Audit()`. L'utilisation de la fonction native est encouragée lorsque c'est possible.

Voir le *tableau d'événements d'audit* pour tous les événements levés par CPython.

Ajouté dans la version 3.8.

`sys.`**`base_exec_prefix`**

Equivalent to `exec_prefix`, but refering to the base Python installation.

When running under *Virtual Environments*, `exec_prefix` gets overwritten to the virtual environment prefix. `base_exec_prefix`, conversely, does not change, and always points to the base Python installation. Refer to *Virtual Environments* for more information.

Ajouté dans la version 3.3.

sys.**base_prefix**

Equivalent to `prefix`, but refering to the base Python installation.

When running under *virtual environment*, `prefix` gets overwritten to the virtual environment prefix. `base_prefix`, conversely, does not change, and always points to the base Python installation. Refer to *Virtual Environments* for more information.

Ajouté dans la version 3.3.

sys.**byteorder**

Un indicateur de l'ordre natif des octets. Vaudra `'big'` sur les plateformes gros-boutistes (octet le plus significatif en premier), et `'little'` sur les plateformes petit-boutiste (octet le moins significatif en premier).

sys.**builtin_module_names**

A tuple of strings containing the names of all modules that are compiled into this Python interpreter. (This information is not available in any other way --- `modules.keys()` only lists the imported modules.)

See also the `sys.stdlib_module_names` list.

sys.**call_tracing**(*func*, *args*)

Call `func(*args)`, while tracing is enabled. The tracing state is saved, and restored afterwards. This is intended to be called from a debugger from a checkpoint, to recursively debug or profile some other code.

Tracing is suspended while calling a tracing function set by `settrace()` or `setprofile()` to avoid infinite recursion. `call_tracing()` enables explicit recursion of the tracing function.

sys.**copyright**

Une chaîne contenant le copyright relatif à l'interpréteur Python.

sys.**_clear_type_cache**()

Vide le cache interne de types. Le cache de types est utilisé pour accélérer les recherches d'attributs et de méthodes. N'utilisez cette fonction *que* pour libérer des références inutiles durant le débogage de fuite de référence.

Cette fonction ne devrait être utilisée que pour un usage interne et spécialisé.

Obsolète depuis la version 3.13 : Use the more general `_clear_internal_caches()` function instead.

sys.**_clear_internal_caches**()

Clear all internal performance-related caches. Use this function *only* to release unnecessary references and memory blocks when hunting for leaks.

Ajouté dans la version 3.13.

sys.**_current_frames**()

Renvoie un dictionnaire faisant correspondre chaque identifiant de fil d'exécution à la *stack frame* actuellement active pour ces fils d'exécution au moment où la fonction est appelée. Notez que les fonctions du module `traceback` peuvent construire une *call stack* à partir d'une telle *frame*.

N'ayant pas besoin de la coopération des fils d'exécution bloqués, cette fonction est très utile pour déboguer un *deadlock*. Aussi, les *call stack* de ces fils d'exécution ne changeront pas tant qu'ils seront bloqués. La *frame* renvoyée pour un fil d'exécution non bloqué peut ne plus être liée à l'activité courante du fil d'exécution au moment où le code appelant examine la *frame*.

Cette fonction ne devrait être utilisée que pour un usage interne et spécialisé.

Lève un *événement d'audit* `sys._current_frames` sans arguments.

sys.**_current_exceptions**()

Return a dictionary mapping each thread's identifier to the topmost exception currently active in that thread at the time the function is called. If a thread is not currently handling an exception, it is not included in the result dictionary.

This is most useful for statistical profiling.

Cette fonction ne devrait être utilisée que pour un usage interne et spécialisé.

Raises an *auditing event* `sys._current_exceptions` with no arguments.

Modifié dans la version 3.12 : Each value in the dictionary is now a single exception instance, rather than a 3-tuple as returned from `sys.exc_info()`.

`sys.breakpointhook()`

Cette fonction auto-déclenchée (*hook function* en anglais) est appelée par la fonction native `breakpoint()`. Par défaut, elle vous place dans le débogueur `pdb`, mais elle peut être dirigée vers n'importe quelle autre fonction pour que vous puissiez choisir le débogueur utilisé.

La signature de cette fonction dépend de ce qu'elle appelle. Par exemple, l'appel par défaut (e.g. `pdb.set_trace()`) n'attend pas d'argument, mais vous pourriez la lier à une fonction qui attend des arguments supplémentaires (positionnels et/ou mots-clés). La fonction native `breakpoint()` passe ses `*args` et `**kws` directement au travers. Tout ce que renvoie `breakpointhooks()` est renvoyé par `breakpoint()`.

L'implémentation par défaut consulte d'abord la variable d'environnement `PYTHONBREAKPOINT`. Si elle vaut `"0"` alors cette fonction s'achève immédiatement (elle ne fait donc rien). Si la variable d'environnement n'est pas définie, ou s'il s'agit d'une chaîne vide, `pdb.set_trace()` est appelée. Sinon cette variable doit nommer une fonction à appeler, en utilisant la syntaxe d'importation de Python, par exemple `package.subpackage.module.function`. Dans ce cas, `package.subpackage.module` sera importé et le module devra contenir une fonction appelable `function()`. Celle-ci est lancée en lui passant `*args` et `*kws` et, quoique renvoie `function()`, `sys.breakpointhook()` retourne à la fonction native `breakpoint()`.

Notez que si un problème apparaît au moment de l'importation de la fonction nommée dans `PYTHONBREAKPOINT`, une alerte `RuntimeWarning` est indiquée et le point d'arrêt est ignoré.

Notez également que si `sys.breakpointhook()` est surchargé de manière programmatique, `PYTHONBREAKPOINT` *n'est pas* consulté.

Ajouté dans la version 3.7.

`sys._debugmallocstats()`

Affiche des informations bas-niveau sur la sortie d'erreur à propos de l'état de l'allocateur de mémoire de CPython.

If Python is built in debug mode (`configure --with-pydebug option`), it also performs some expensive internal consistency checks.

Ajouté dans la version 3.3.

Cette fonction est spécifique à CPython. Le format de sa sortie n'est pas définit ici et pourrait changer.

`sys.dllhandle`

Nombre entier spécifiant le descripteur de la DLL Python.

Availability : Windows.

`sys.displayhook(value)`

Si *value* n'est pas `None`, cette fonction écrit `repr(value)` sur `sys.stdout`, et sauvegarde *value* dans `builtins._`. Si `repr(value)` ne peut pas être encodé avec `sys.stdout.encoding` en utilisant le gestionnaire d'erreur `sys.stdout.errors` (qui est probablement `'strict'`), elle sera encodée avec `sys.stdout.encoding` en utilisant le gestionnaire d'erreur `'backslashreplace'`.

`sys.displayhook` est appelé avec le résultat de l'évaluation d'une *expression* entrée dans une session Python interactive. L'affichage de ces valeurs peut être personnalisé en assignant une autre fonction d'un argument à `sys.displayhook`.

Pseudo-code :

```
def displayhook(value):
    if value is None:
        return
    # Set '_' to None to avoid recursion
```

(suite sur la page suivante)

(suite de la page précédente)

```
    builtins._ = None
    text = repr(value)
    try:
        sys.stdout.write(text)
    except UnicodeEncodeError:
        bytes = text.encode(sys.stdout.encoding, 'backslashreplace')
        if hasattr(sys.stdout, 'buffer'):
            sys.stdout.buffer.write(bytes)
        else:
            text = bytes.decode(sys.stdout.encoding, 'strict')
            sys.stdout.write(text)
    sys.stdout.write("\n")
    builtins._ = value
```

Modifié dans la version 3.2 : Utiliser le gestionnaire d'erreur `'backslashreplace'` en cas d'*UnicodeEncodeError*.

`sys.`**`dont_write_bytecode`**

Si vrai, Python n'essaiera pas d'écrire de fichiers `.pyc` à l'importation de modules source. Cette valeur est initialement définie à `True` ou `False` en fonction de l'option de la ligne de commande `-B` et de la variable d'environnement `PYTHONDONDONTWRITEBYTECODE`, mais vous pouvez aussi la modifier vous-même pour contrôler la génération des fichiers de *bytecode*.

`sys.`**`_emscripten_info`**

A *named tuple* holding information about the environment on the *wasm32-emscripten* platform. The named tuple is provisional and may change in the future.

`_emscripten_info.`**`emscripten_version`**

Emscripten version as tuple of ints (major, minor, micro), e.g. `(3, 1, 8)`.

`_emscripten_info.`**`runtime`**

Runtime string, e.g. browser user agent, `'Node.js v14.18.2'`, or `'UNKNOWN'`.

`_emscripten_info.`**`pthreads`**

`True` if Python is compiled with Emscripten pthreads support.

`_emscripten_info.`**`shared_memory`**

`True` if Python is compiled with shared memory support.

Availability : Emscripten.

Ajouté dans la version 3.11.

`sys.`**`pycache_prefix`**

If this is set (not `None`), Python will write bytecode-cache `.pyc` files to (and read them from) a parallel directory tree rooted at this directory, rather than from `__pycache__` directories in the source code tree. Any `__pycache__` directories in the source code tree will be ignored and new `.pyc` files written within the pycache prefix. Thus if you use *compileall* as a pre-build step, you must ensure you run it with the same pycache prefix (if any) that you will use at runtime.

Un chemin relatif est interprété relativement au répertoire courant.

Cette valeur est initialement définie basée sur la valeur de l'option de ligne de commande `-X pycache_prefix=PATH` ou la variable d'environnement `PYTHONPYCACHEPREFIX` (La ligne de commande est prioritaire). Si aucune des deux options n'est définie, alors la valeur est `None`.

Ajouté dans la version 3.8.

`sys.`**`excepthook`**(*type*, *value*, *traceback*)

Cette fonction affiche la *traceback* et l'exception donnée sur `sys.stderr`.

When an exception other than *SystemExit* is raised and uncaught, the interpreter calls `sys.excepthook`

with three arguments, the exception class, exception instance, and a traceback object. In an interactive session this happens just before control is returned to the prompt ; in a Python program this happens just before the program exits. The handling of such top-level exceptions can be customized by assigning another three-argument function to `sys.excepthook`.

Lève un événement d'audit `sys.excepthook` avec les arguments `hook`, `type`, `value`, `traceback` lorsqu'une exception non interceptée se produit. Si aucun point d'entrée n'a été défini, `hook` peut être `None`. Si n'importe quel point d'entrée lève une exception dérivée de *`RuntimeError`*, l'appel au point d'entrée sera supprimé. Autrement, l'exception du point d'entrée d'audit sera reporté comme *non levable* (**unraisable** en anglais) et `sys.excepthook` sera appelé.

> **Voir aussi**
>
> La fonction *`sys.unraisablehook()`* gère les exceptions *non-levables* et la fonction *`threading.excepthook()`* gère les exceptions levées par *`threading.Thread.run()`*.

`sys.`**`__breakpointhook__`**

`sys.`**`__displayhook__`**

`sys.`**`__excepthook__`**

`sys.`**`__unraisablehook__`**

Ces objets contiennent les valeurs originales de `breakpointhook`, `displayhook`, `excepthook` et `unraisablehook` au début du programme.Ils sont sauvegardés de façon à ce que `breakpointhook`, `displayhook`, `excepthook` et `unraisablehook` puissent être restaurés au cas où ils seraient remplacés par des objets cassés ou alternatifs.

Ajouté dans la version 3.7 : `__breakpointhook__`

Ajouté dans la version 3.8 : `__unraisablehook__`

`sys.`**`exception`**`()`

This function, when called while an exception handler is executing (such as an `except` or `except*` clause), returns the exception instance that was caught by this handler. When exception handlers are nested within one another, only the exception handled by the innermost handler is accessible.

If no exception handler is executing, this function returns `None`.

Ajouté dans la version 3.11.

`sys.`**`exc_info`**`()`

This function returns the old-style representation of the handled exception. If an exception `e` is currently handled (so *`exception()`* would return `e`), *`exc_info()`* returns the tuple `(type(e), e, e.__traceback__)`. That is, a tuple containing the type of the exception (a subclass of *`BaseException`*), the exception itself, and a traceback object which typically encapsulates the call stack at the point where the exception last occurred.

If no exception is being handled anywhere on the stack, this function return a tuple containing three `None` values.

Modifié dans la version 3.11 : The `type` and `traceback` fields are now derived from the `value` (the exception instance), so when an exception is modified while it is being handled, the changes are reflected in the results of subsequent calls to *`exc_info()`*.

`sys.`**`exec_prefix`**

Une chaîne donnant le préfixe de dossier spécifique au site où les fichiers dépendant de la plateforme sont installés. Par défaut, c'est `'/usr/local'`. C'est configurable à la compilation avec l'option `--exec-prefix` du script **`configure`**. Tous les fichiers de configurations (tel que `pyconfig,h`) sont installés dans le dossier *`exec_prefix`*`/lib/python`*`X.Y`*`/config`, et les modules sous forme de bibliothèques partagées sont installés dans *`exec_prefix`*`/lib/python`*`X.Y`*`/lib-dynload`, où *X.Y* est le numéro de version de Python, par exemple `3.2`.

> **Note**
>
> If a *virtual environment* is in effect, this `exec_prefix` will point to the virtual environment. The value for the Python installation will still be available, via `base_exec_prefix`. Refer to *Virtual Environments* for more information.

Modifié dans la version 3.14 : When running under a *virtual environment*, `prefix` and `exec_prefix` are now set to the virtual environment prefix by the *path initialization*, instead of `site`. This means that `prefix` and `exec_prefix` always point to the virtual environment, even when `site` is disabled (`-S`).

`sys.`**`executable`**

Une chaîne donnant le chemin absolu vers l'interpréteur Python, un fichier binaire exécutable, sur les système sur lesquels ça à du sens. Si Python n'est pas capable de récupérer le chemin réel de son exécutable, `sys.executable` sera une chaîne vide ou `None`.

`sys.`**`exit`**`(`[*arg*]`)`

Raise a `SystemExit` exception, signaling an intention to exit the interpreter.

L'argument optionnel *arg* peut être un nombre entier donnant l'état de sortie (zéro par défaut), ou un autre type d'objet. Pour les *shells* (et autres), si c'est un entier, zéro signifie "terminé avec succès", et toutes les autres valeurs signifient "terminé anormalement". La plupart des systèmes imposent qu'il se situe dans la plage 0--127, et leur comportement n'est pas défini pour les autres cas. Certains systèmes peu communs ont pour convention d'assigner un sens particulier à des valeur spécifiques. Les programmes Unix utilisent généralement 2 pour les erreurs de syntaxe dans les arguments de la ligne de commande, et 1 pour toutes les autres erreurs. Si un autre type est passé, `None` est équivalent à zéro, et tout autre objet est écrit sur `stderr` et donne un code de sortie 1. Typiquement, `sys.exit("some error message")` est un moyen rapide de quitter un programme en cas d'erreur.

Since `exit()` ultimately "only" raises an exception, it will only exit the process when called from the main thread, and the exception is not intercepted. Cleanup actions specified by finally clauses of `try` statements are honored, and it is possible to intercept the exit attempt at an outer level.

Modifié dans la version 3.6 : Si une erreur survient lors du nettoyage après que l'interpréteur Python ait intercepté un `SystemExit` (typiquement une erreur en vidant les tampons des sorties standard), le code de sortie est changé à 120.

`sys.`**`flags`**

La *named tuple flags* expose l'état des options de ligne de commande. Ces attributs sont en lecture seule.

flags.**debug**	-d
flags.**inspect**	-i
flags.**interactive**	-i
flags.**isolated**	-I
flags.**optimize**	-O or -OO
flags.**dont_write_bytecode**	-B
flags.**no_user_site**	-s
flags.**no_site**	-S
flags.**ignore_environment**	-E
flags.**verbose**	-v
flags.**bytes_warning**	-b
flags.**quiet**	-q
flags.**hash_randomization**	-R
flags.**dev_mode**	-X dev (*Python en mode développement*)
flags.**utf8_mode**	-X utf8
flags.**safe_path**	-P
flags.**int_max_str_digits**	-X int_max_str_digits (*integer string conversion length limitation*)
flags.**warn_default_encoding**	-X warn_default_encoding

Modifié dans la version 3.2 : Ajout de l'attribut `quiet` pour la nouvelle option `-q`.

Ajouté dans la version 3.2.3 : L'attribut `hash_randomization`.

Modifié dans la version 3.3 : Suppression de l'attribut obsolète `division_warning`.

Modifié dans la version 3.4 : Ajout de l'attribut `isolated` pour l'option `-I isolated`.

Modifié dans la version 3.7 : Ajout de l'attribut `dev_mode` pour le nouveau *Mode Développeur Python* et l'attribut `utf8_mode` pour la nouvelle option `-X utf8`.

Modifié dans la version 3.10 : Added `warn_default_encoding` attribute for `-X warn_default_encoding` flag.

Modifié dans la version 3.11 : Added the `safe_path` attribute for `-P` option.

Modifié dans la version 3.11 : Added the `int_max_str_digits` attribute.

`sys.`**`float_info`**

Un *named tuple* contenant des informations à propos du type *float*. Il contient des informations de bas niveau à propos de la précision et de la représentation interne. Les valeurs correspondent aux différentes constantes à propos des nombres à virgule flottantes définies dans le fichier d'entête `float.h` pour le langage de programmation C. Voir la section 5.2.4.2.2 de *1999 ISO/IEC C standard* [C99], *Characteristics of floating types*, pour plus de détails.

Tableau 1 – Attributes of the `float_info` named tuple

attribut	macro *float.h*	explication
`float_info.`**`epsilon`**	`DBL_EPSILON`	difference between 1.0 and the least value greater than 1.0 that is representable as a float. Voir aussi : `math.ulp()`.
`float_info.`**`dig`**	`DBL_DIG`	The maximum number of decimal digits that can be faithfully represented in a float ; see below.
`float_info.`**`mant_dig`**	`DBL_MANT_DIG`	Float precision : the number of base-`radix` digits in the significand of a float.
`float_info.`**`max`**	`DBL_MAX`	The maximum representable positive finite float.
`float_info.`**`max_exp`**	`DBL_MAX_EXP`	The maximum integer *e* such that `radix**(e-1)` is a representable finite float.
`float_info.`**`max_10_exp`**	`DBL_MAX_10_EXP`	The maximum integer *e* such that `10**e` is in the range of representable finite floats.
`float_info.`**`min`**	`DBL_MIN`	The minimum representable positive *normalized* float. Utilisez `math.ulp(0.0)` pour obtenir le plus petit nombre a virgule positif *dénormalisé* représentable.
`float_info.`**`min_exp`**	`DBL_MIN_EXP`	The minimum integer *e* such that `radix**(e-1)` is a normalized float.
`float_info.`**`min_10_exp`**	`DBL_MIN_10_EXP`	The minimum integer *e* such that `10**e` is a normalized float.
`float_info.`**`radix`**	`FLT_RADIX`	The radix of exponent representation.
`float_info.`**`rounds`**	`FLT_ROUNDS`	An integer representing the rounding mode for floating-point arithmetic. This reflects the value of the system `FLT_ROUNDS` macro at interpreter startup time : — `-1` : indeterminable — `0` : toward zero — `1` : to nearest — `2` : toward positive infinity — `3` : toward negative infinity All other values for `FLT_ROUNDS` characterize implementation-defined rounding behavior.

The attribute `sys.float_info.dig` needs further explanation. If `s` is any string representing a decimal number with at most `sys.float_info.dig` significant digits, then converting `s` to a float and back again will recover a string representing the same decimal value :

```
>>> import sys
>>> sys.float_info.dig
```

(suite sur la page suivante)

(suite de la page précédente)

```
15
>>> s = '3.14159265358979'    # decimal string with 15 significant digits
>>> format(float(s), '.15g')  # convert to float and back -> same value
'3.14159265358979'
```

Cependant, pour les chaînes avec plus de `sys.float_info.dig` chiffres significatifs, ce n'est pas toujours vrai :

```
>>> s = '9876543211234567'    # 16 significant digits is too many!
>>> format(float(s), '.16g')  # conversion changes value
'9876543211234568'
```

`sys.float_repr_style`

Une chaîne indiquant comment la fonction `repr()` se comporte avec les nombres à virgule flottante. Si la chaîne a la valeur `'short'`, alors pour un *float* finit `x`, `repr(x)` essaye de donner une courte chaîne tel que `float(repr(x)) == x`. C'est le comportement typique à partir de Python 3.1. Autrement, `float_repr_style` a la valeur `'legacy'` et `repr(x)` se comporte comme les versions antérieures à 3.1.

Ajouté dans la version 3.1.

`sys.getallocatedblocks()`

Return the number of memory blocks currently allocated by the interpreter, regardless of their size. This function is mainly useful for tracking and debugging memory leaks. Because of the interpreter's internal caches, the result can vary from call to call ; you may have to call `_clear_internal_caches()` and `gc.collect()` to get more predictable results.

If a Python build or implementation cannot reasonably compute this information, `getallocatedblocks()` is allowed to return 0 instead.

Ajouté dans la version 3.4.

`sys.getunicodeinternedsize()`

Return the number of unicode objects that have been interned.

Ajouté dans la version 3.12.

`sys.getandroidapilevel()`

Return the build-time API level of Android as an integer. This represents the minimum version of Android this build of Python can run on. For runtime version information, see `platform.android_ver()`.

Availability : Android.

Ajouté dans la version 3.7.

`sys.getdefaultencoding()`

Renvoie le nom du codage par défaut actuellement utilisé par l'implémentation *Unicode* pour coder les chaînes.

`sys.getdlopenflags()`

Return the current value of the flags that are used for `dlopen()` calls. Symbolic names for the flag values can be found in the `os` module (`RTLD_xxx` constants, e.g. `os.RTLD_LAZY`).

Availability : Unix.

`sys.getfilesystemencoding()`

Get the *filesystem encoding* : the encoding used with the *filesystem error handler* to convert between Unicode filenames and bytes filenames. The filesystem error handler is returned from `getfilesystemencodeerrors()`.

For best compatibility, str should be used for filenames in all cases, although representing filenames as bytes is also supported. Functions accepting or returning filenames should support either str or bytes and internally convert to the system's preferred representation.

Les fonctions `os.fsencode()` et `os.fsdecode()` devraient être utilisées pour s'assurer qu'un encodage et un gestionnaire d'erreurs correct sont utilisés.

L'*encodage du système de fichiers et gestionnaire d'erreurs associé* sont configurés au démarrage de Python par la fonction `PyConfig_Read()` : regardez `filesystem_encoding` et `filesystem_errors` dans les membres de `PyConfig`.

Modifié dans la version 3.2 : `getfilesystemencoding()` ne peut plus renvoyer `None`.

Modifié dans la version 3.6 : Sur Windows, on est plus assurés d'obtenir `'mbcs'`. Voir la **PEP 529** et `_enablelegacywindowsfsencoding()` pour plus d'informations.

Modifié dans la version 3.7 : Return `'utf-8'` if the *Python UTF-8 Mode* is enabled.

`sys.`**`getfilesystemencodeerrors`**`()`

Get the *filesystem error handler* : the error handler used with the *filesystem encoding* to convert between Unicode filenames and bytes filenames. The filesystem encoding is returned from `getfilesystemencoding()`.

Les fonctions `os.fsencode()` et `os.fsdecode()` devraient être utilisées pour s'assurer qu'un encodage et un gestionnaire d'erreurs correct sont utilisés.

L'*encodage du système de fichiers et gestionnaire d'erreurs associé* sont configurés au démarrage de Python par la fonction `PyConfig_Read()` : regardez `filesystem_encoding` et `filesystem_errors` dans les membres de `PyConfig`.

Ajouté dans la version 3.6.

`sys.`**`get_int_max_str_digits`**`()`

Returns the current value for the *integer string conversion length limitation*. See also `set_int_max_str_digits()`.

Ajouté dans la version 3.11.

`sys.`**`getrefcount`**`(`*object*`)`

Donne le nombre de référence de l'objet *object*. Le nombre renvoyé est généralement d'une référence de plus qu'attendu, puisqu'il compte la référence (temporaire) de l'argument à `getrefcount()`.

Note that the returned value may not actually reflect how many references to the object are actually held. For example, some objects are *immortal* and have a very high refcount that does not reflect the actual number of references. Consequently, do not rely on the returned value to be accurate, other than a value of 0 or 1.

Particularité de l'implémentation CPython : *Immortal* objects with a large reference count can be identified via `_is_immortal()`.

Modifié dans la version 3.12 : Immortal objects have very large refcounts that do not match the actual number of references to the object.

`sys.`**`getrecursionlimit`**`()`

Donne la limite actuelle de la limite de récursion, la profondeur maximum de la pile de l'interpréteur. Cette limite empêche Python de planter lors d'une récursion infinie à cause d'un débordement de la pile. Elle peut être modifiée par `setrecursionlimit()`.

`sys.`**`getsizeof`**`(`*object*[, *default*]`)`

Donne la taille d'un objet en octets. L'objet peut être de n'importe quel type. Le résultat sera correct pour tous les objets natifs, mais le résultat peut ne pas être toujours vrai pour les extensions, la valeur étant dépendante de l'implémentation.

Seule la mémoire directement attribuée à l'objet est prise en compte, pas la mémoire consommée par les objets vers lesquels il a des références.

S'il est fourni, *default* sera renvoyé si l'objet ne fournit aucun moyen de récupérer sa taille. Sinon, une exception `TypeError` sera levée.

`getsizeof()` appelle la méthode `__sizeof__` de l'objet, et s'il est géré par lui, ajoute le surcoût du ramasse-miettes.

See recursive sizeof recipe for an example of using `getsizeof()` recursively to find the size of containers and all their contents.

`sys.getswitchinterval()`

Renvoie la valeur du *thread switch interval* de l'interpréteur, voir `setswitchinterval()`.

Ajouté dans la version 3.2.

`sys._getframe([depth])`

Renvoie une *frame* de la pile d'appels. Si le nombre entier optionnel *depth* est donné, la *frame* donnée sera de *depth* appels depuis le haut de la pile. Si c'est plus profond que la hauteur de la pile, une exception `ValueError` est levée. La profondeur par défaut est zéro, donnant ainsi la *frame* du dessus de la pile.

Raises an *auditing event* `sys._getframe` with argument `frame`.

Cette fonction ne devrait être utilisée que pour une utilisation interne et spécifique. Il n'est pas garanti qu'elle existe dans toutes les implémentations de Python.

`sys._getframemodulename([depth])`

Return the name of a module from the call stack. If optional integer *depth* is given, return the module that many calls below the top of the stack. If that is deeper than the call stack, or if the module is unidentifiable, `None` is returned. The default for *depth* is zero, returning the module at the top of the call stack.

Raises an *auditing event* `sys._getframemodulename` with argument `depth`.

Cette fonction ne devrait être utilisée que pour une utilisation interne et spécifique. Il n'est pas garanti qu'elle existe dans toutes les implémentations de Python.

`sys.getobjects(limit[, type])`

This function only exists if CPython was built using the specialized configure option `--with-trace-refs`. It is intended only for debugging garbage-collection issues.

Return a list of up to *limit* dynamically allocated Python objects. If *type* is given, only objects of that exact type (not subtypes) are included.

Objects from the list are not safe to use. Specifically, the result will include objects from all interpreters that share their object allocator state (that is, ones created with `PyInterpreterConfig.use_main_obmalloc` set to 1 or using `Py_NewInterpreter()`, and the main interpreter). Mixing objects from different interpreters may lead to crashes or other unexpected behavior.

Particularité de l'implémentation CPython : This function should be used for specialized purposes only. It is not guaranteed to exist in all implementations of Python.

Modifié dans la version 3.14 : The result may include objects from other interpreters.

`sys.getprofile()`

Renvoie la fonction de profilage tel que défini par `setprofile()`.

`sys.gettrace()`

Renvoie la fonction de traçage tel que définie par `settrace()`.

La fonction `gettrace()` ne sert que pour implémenter des débogueurs, des *profilers*, outils d'analyse de couverture, etc…. Son comportement dépend de l'implémentation et non du langage, elle n'est donc pas forcément disponible dans toutes les implémentations de Python.

`sys.getwindowsversion()`

Renvoie un *n*-uplet nommé décrivant la version de Windows en cours d'exécution. Les attributs nommés sont *major*, *minor*, *build*, *platform*, *service_pack*, *service_pack_minor*, *service_pack_major*, *suite_mask*, *product_type* et *platform_version*. *service_pack* contient une string, *platform_version* un triplet, et tous les autres sont des nombres entiers. Ces attributs sont également accessibles par leur nom, donc `sys.getwindowsversion()[0]` est équivalent à `sys.getwindowsversion().major`. Pour des raisons de compatibilité avec les versions antérieures, seuls les 5 premiers éléments sont accessibles par leur indice.

platform will be `2` (VER_PLATFORM_WIN32_NT).

product_type peut être une des valeurs suivantes :

Constante	Signification
1 (VER_NT_WORKSTATION)	Le système une station de travail.
2 (VER_NT_DOMAIN_CONTROLLER)	Le système est un contrôleur de domaine.
3 (VER_NT_SERVER)	Le système est un serveur, mais pas un contrôleur de domaine.

This function wraps the Win32 `GetVersionEx()` function; see the Microsoft documentation on `OSVERSIONINFOEX()` for more information about these fields.

platform_version returns the major version, minor version and build number of the current operating system, rather than the version that is being emulated for the process. It is intended for use in logging rather than for feature detection.

> **Note**
>
> *platform_version* derives the version from kernel32.dll which can be of a different version than the OS version. Please use `platform` module for achieving accurate OS version.

Availability : Windows.

Modifié dans la version 3.2 : Changé en un *n*-uplet nommé, et ajout de *service_pack_minor*, *service_pack_major*, *suite_mask*, et *product_type*.

Modifié dans la version 3.6 : Ajout de *platform_version*

`sys.`**`get_asyncgen_hooks`**`()`

Renvoie un objet *asyncgen_hooks*, qui est semblable à un `namedtuple` de la forme `(firstiter, finalizer)`, où *firstiter* et *finalizer* sont soit `None` ou des fonctions qui prennent un *asynchronous generator iterator* comme argument, et sont utilisées pour planifier la finalisation d'un générateur asynchrone par un *event loop*.

Ajouté dans la version 3.6 : Voir la **PEP 525** pour plus d'informations.

> **Note**
>
> Cette fonction à été ajoutée à titre provisoire (voir la **PEP 411** pour plus d'informations.)

`sys.`**`get_coroutine_origin_tracking_depth`**`()`

Récupère le nombre de cadres d'exécution conservés par les coroutines pour le suivi de leur création, telle que défini par `set_coroutine_origin_tracking_depth()`.

Ajouté dans la version 3.7.

> **Note**
>
> Cette fonction à été ajoutée à titre provisoire (Voir la **PEP 411** pour plus d'informations.) Utilisez la uniquement à des fins de débogage.

`sys.`**`hash_info`**

Un *named tuple* donnant les paramètres de l'implémentation de la fonction de hachage de nombres. Pour plus d'informations sur le hachage des types numériques, consultez *Hachage des types numériques*.

`hash_info.`**`width`**

The width in bits used for hash values

`hash_info.`**`modulus`**

The prime modulus P used for numeric hash scheme

`hash_info.inf`
The hash value returned for a positive infinity

`hash_info.nan`
(This attribute is no longer used)

`hash_info.imag`
The multiplier used for the imaginary part of a complex number

`hash_info.algorithm`
The name of the algorithm for hashing of str, bytes, and memoryview

`hash_info.hash_bits`
The internal output size of the hash algorithm

`hash_info.seed_bits`
The size of the seed key of the hash algorithm

Ajouté dans la version 3.2.

Modifié dans la version 3.4 : Ajout de *algorithm*, *hash_bits* et *seed_bits*

`sys.hexversion`
Le numéro de version codé sous forme d'un seul nombre entier. Ce numéro augmente avec chaque version, y compris pour les versions hors production. Par exemple, pour vérifier que l'interpréteur Python est au moins la version 1.5, utilisez :

```
if sys.hexversion >= 0x010502F0:
    # use some advanced feature
    ...
else:
    # use an alternative implementation or warn the user
    ...
```

Cet attribut s'appelle `hexversion` dans le sens où il ne semble avoir du sens que s'il est regardé après avoir été passé à la fonction native `hex()`. Le *named tuple* `sys.version_info` représente la même information d'une manière plus humaine.

Consultez apiabiversion pour plus d'informations sur `hexversion`.

`sys.implementation`
Un objet contenant des informations sur l'implémentation de la version actuelle de l'interpréteur Python. Les attributs suivants existent obligatoirement sur toutes les implémentations Python.

name est l'identifiant de l'implémentation, e.g. `'cpython'`. Cette chaîne est définie par l'implémentation de Python, mais sera toujours en minuscule.

version est un *n*-uplet nommé, du même format que `sys.version_info`. Il représente la version de l'**implementation** de Python. C'est une information différente de la version du **langage** auquel l'interpréteur actuel se conforme (donnée par `sys.version_info`). Par exemple, pour PyPy 1.8 `sys.implementation.version` peut valoir `sys.version_info(1, 8, 0, 'final', 0)`, alors que `sys.version_info` peut valoir `sys.version_info(2, 7, 2, 'final', 0)`. Pour CPython ces deux valeurs sont identiques puisque c'est l'implémentation de référence.

hexversion est la version de l'implémentation sous forme hexadécimale, comme `sys.hexversion`.

cache_tag est la balise utilisée par le mécanisme d'importation dans les noms de fichiers des modules mis en cache. Par convention, il devrait se composer du nom et de la version de l'implémentation, comme `'cpython-33'`. Cependant, une implémentation Python peut utiliser une autre valeur si nécessaire. `cache_tag` à `None` signifie que la mise en cache des modules doit être désactivée.

`sys.implementation` peut contenir d'autres attributs spécifiques à l'implémentation de Python. Ces attributs spécifiques doivent commencer par un *underscore*, et ne sont pas documentés ici. Indépendamment de son contenu, `sys.implementation` ne change jamais durant l'exécution de l'interpréteur, ni entre les versions

d'une même implémentation. (Il peut cependant changer entre les versions du langage Python.) Voir la **PEP 421** pour plus d'informations.

Ajouté dans la version 3.3.

> **Note**
>
> L'addition de nouveaux attributs requis doivent passer par le processus de **PEP** classique. Voir **PEP 421** pour plus d'informations.

`sys.int_info`

Un *named tuple* qui contient des informations sur la représentation interne des entiers de Python. Les attributs sont en lecture seule.

`int_info.bits_per_digit`

The number of bits held in each digit. Python integers are stored internally in base `2**int_info.bits_per_digit`.

`int_info.sizeof_digit`

The size in bytes of the C type used to represent a digit.

`int_info.default_max_str_digits`

The default value for `sys.get_int_max_str_digits()` when it is not otherwise explicitly configured.

`int_info.str_digits_check_threshold`

The minimum non-zero value for `sys.set_int_max_str_digits()`, `PYTHONINTMAXSTRDIGITS`, or `-X int_max_str_digits`.

Ajouté dans la version 3.1.

Modifié dans la version 3.11 : Added `default_max_str_digits` and `str_digits_check_threshold`.

`sys.__interactivehook__`

Lorsque cet attribut existe, sa valeur est automatiquement appelée (sans argument) par l'interpréteur lors de son démarrage en mode interactif. L'appel se fait après que le fichier `PYTHONSTARTUP` soit lu, afin que vous puissiez y configurer votre fonction. *Configuré* par le module `site`.

Lève un *évènement d'audit* `cpython.run_interactivehook` avec l'objet de point d'entrée comme argument lorsqu'il est appelé au démarrage.

Ajouté dans la version 3.4.

`sys.intern(string)`

Ajoute *string* dans le tableau des chaînes "internées" et renvoie la chaîne internée -- qui peut être *string* elle-même ou une copie. Interner une chaîne de caractères permet de gagner un peu de performance lors de l'accès aux dictionnaires -- si les clés du dictionnaire et la clé recherchée sont internées, les comparaisons de clés (après le hachage) pourront se faire en comparant les pointeurs plutôt que caractère par caractère. Normalement, les noms utilisés dans les programmes Python sont automatiquement internés, et les dictionnaires utilisés pour stocker les attributs de modules, de classes, ou d'instances ont aussi leurs clés internées.

Interned strings are not *immortal* ; you must keep a reference to the return value of `intern()` around to benefit from it.

`sys._is_gil_enabled()`

Return `True` if the *GIL* is enabled and `False` if it is disabled.

Ajouté dans la version 3.13.

`sys.is_finalizing()`

Return `True` if the main Python interpreter is *shutting down*. Return `False` otherwise.

See also the `PythonFinalizationError` exception.

Ajouté dans la version 3.5.

sys.**last_exc**

This variable is not always defined ; it is set to the exception instance when an exception is not handled and the interpreter prints an error message and a stack traceback. Its intended use is to allow an interactive user to import a debugger module and engage in post-mortem debugging without having to re-execute the command that caused the error. (Typical use is `import pdb; pdb.pm()` to enter the post-mortem debugger ; see *pdb* module for more information.)

Ajouté dans la version 3.12.

sys.**_is_immortal**(*op*)

Return `True` if the given object is *immortal*, `False` otherwise.

> **Note**
>
> Objects that are immortal (and thus return `True` upon being passed to this function) are not guaranteed to be immortal in future versions, and vice versa for mortal objects.

Ajouté dans la version 3.14.

Particularité de l'implémentation CPython : This function should be used for specialized purposes only. It is not guaranteed to exist in all implementations of Python.

sys.**_is_interned**(*string*)

Return `True` if the given string is "interned", `False` otherwise.

Ajouté dans la version 3.13.

Particularité de l'implémentation CPython : It is not guaranteed to exist in all implementations of Python.

sys.**last_type**

sys.**last_value**

sys.**last_traceback**

These three variables are deprecated ; use `sys.last_exc` instead. They hold the legacy representation of `sys.last_exc`, as returned from `exc_info()` above.

sys.**maxsize**

Un entier donnant à la valeur maximale qu'une variable de type `Py_ssize_t` peut prendre. C'est typiquement `2**31 - 1` sur une plateforme 32 bits et `2**63 - 1` sur une plateforme 64 bits.

sys.**maxunicode**

Un entier donnant la valeur du plus grand point de code Unicode, c'est-à-dire `1114111` (`0x10FFFF` en hexadécimal).

Modifié dans la version 3.3 : Avant la **PEP 393**, `sys.maxunicode` valait soit `0xFFFF` soit `0x10FFFF`, en fonction l'option de configuration qui spécifiait si les caractères Unicode étaient stockés en UCS-2 ou UCS-4.

sys.**meta_path**

A list of *meta path finder* objects that have their `find_spec()` methods called to see if one of the objects can find the module to be imported. By default, it holds entries that implement Python's default import semantics. The `find_spec()` method is called with at least the absolute name of the module being imported. If the module to be imported is contained in a package, then the parent package's `__path__` attribute is passed in as a second argument. The method returns a *module spec*, or `None` if the module cannot be found.

> **Voir aussi**
>
> ***importlib.abc.MetaPathFinder***
> La classe de base abstraite définissant l'interface des objets *finder* de `meta_path`.

> ***importlib.machinery.ModuleSpec***
> La classe concrète dont `find_spec()` devrait renvoyer des instances.

Modifié dans la version 3.4 : *Module specs* were introduced in Python 3.4, by **PEP 451**.

Modifié dans la version 3.12 : Removed the fallback that looked for a `find_module()` method if a `meta_path` entry didn't have a `find_spec()` method.

`sys.`**`modules`**

This is a dictionary that maps module names to modules which have already been loaded. This can be manipulated to force reloading of modules and other tricks. However, replacing the dictionary will not necessarily work as expected and deleting essential items from the dictionary may cause Python to fail. If you want to iterate over this global dictionary always use `sys.modules.copy()` or `tuple(sys.modules)` to avoid exceptions as its size may change during iteration as a side effect of code or activity in other threads.

`sys.`**`orig_argv`**

The list of the original command line arguments passed to the Python executable.

The elements of `sys.orig_argv` are the arguments to the Python interpreter, while the elements of `sys.argv` are the arguments to the user's program. Arguments consumed by the interpreter itself will be present in `sys.orig_argv` and missing from `sys.argv`.

Ajouté dans la version 3.10.

`sys.`**`path`**

Une liste de chaînes de caractères spécifiant les chemins de recherche des modules, initialisée à partir de la variable d'environnement `PYTHONPATH` et d'une valeur par défaut dépendante de l'installation.

By default, as initialized upon program startup, a potentially unsafe path is prepended to `sys.path` (*before* the entries inserted as a result of `PYTHONPATH`) :

— `python -m module` command line : prepend the current working directory.
— `python script.py` command line : prepend the script's directory. If it's a symbolic link, resolve symbolic links.
— `python -c code` and `python` (REPL) command lines : prepend an empty string, which means the current working directory.

To not prepend this potentially unsafe path, use the `-P` command line option or the `PYTHONSAFEPATH` environment variable.

A program is free to modify this list for its own purposes. Only strings should be added to `sys.path`; all other data types are ignored during import.

> **Voir aussi**
>
> — Le module `site` décrit comment utiliser les fichiers *.pth* pour étendre `sys.path`.

`sys.`**`path_hooks`**

Une liste d'appelables d'un argument, *path*, pour essayer de créer un *finder* pour ce chemin. Si un *finder* peut être créé, il doit être renvoyé par l'appelable, sinon une `ImportError` doit être levée.

Précisé à l'origine dans la **PEP 302**.

`sys.`**`path_importer_cache`**

Un dictionnaire faisant office de cache pour les objets *finder*. Les clés sont les chemins qui ont été passés à `sys.path_hooks` et les valeurs sont les *finders* trouvés. Si un chemin est valide selon le système de fichiers mais qu'aucun *finder* n'est trouvé dans `sys.path_hooks`, `None` est stocké.

Précisé à l'origine dans la **PEP 302**.

`sys.`**`platform`**

A string containing a platform identifier. Known values are :

Système	Valeur pour `plateforme`
AIX	`'aix'`
Android	`'android'`
Emscripten	`'emscripten'`
FreeBSD	`'freebsd'`
iOS	`'ios'`
Linux	`'linux'`
macOS	`'darwin'`
Windows	`'win32'`
Windows/Cygwin	`'cygwin'`
WASI	`'wasi'`

On Unix systems not listed in the table, the value is the lowercased OS name as returned by `uname -s`, with the first part of the version as returned by `uname -r` appended, e.g. `'sunos5'`, *at the time when Python was built*. Unless you want to test for a specific system version, it is therefore recommended to use the following idiom :

```
if sys.platform.startswith('sunos'):
    # SunOS-specific code here...
```

Modifié dans la version 3.3 : On Linux, *sys.platform* doesn't contain the major version anymore. It is always `'linux'`, instead of `'linux2'` or `'linux3'`.

Modifié dans la version 3.8 : On AIX, *sys.platform* doesn't contain the major version anymore. It is always `'aix'`, instead of `'aix5'` or `'aix7'`.

Modifié dans la version 3.13 : On Android, *sys.platform* now returns `'android'` rather than `'linux'`.

Modifié dans la version 3.14 : On FreeBSD, *sys.platform* doesn't contain the major version anymore. It is always `'freebsd'`, instead of `'freebsd13'` or `'freebsd14'`.

Voir aussi

os.name has a coarser granularity. *os.uname()* gives system-dependent version information.

Le module *platform* fournit des vérifications détaillées pour l'identité du système.

`sys.`**`platlibdir`**

Nom du dossier de la bibliothèque spécifique à la plateforme. Il est utilisé pour construire le chemin de vers la bibliothèque standard et les chemins vers les modules d'extensions installés.

C'est égal à `"lib"` sur la plupart des plateformes. Sur Fedora et SuSE, c'est égal à `"lib64"` sur les plateformes 64-bits qui renvoient les chemins `sys.path` suivants (où `X.Y` et la version `majeure.mineur` de Python) :

— `/usr/lib64/pythonX.Y/` : Bibliothèque standard (comme `os.py` du module *os*)
— `/usr/lib64/pythonX.Y/lib-dynload/` : Modules d'extension C de la bibliothèque standard (comme le module *errno*, le nom du fichier exact est spécifique à la plateforme)
— `/usr/lib/pythonX.Y/site-packages/` (toujours utiliser `lib`, et non *sys.platlibdir*) : modules tiers
— `/usr/lib64/pythonX.Y/site-packages/` : Modules d'extension C de paquets tiers

Ajouté dans la version 3.9.

`sys.`**`prefix`**

A string giving the site-specific directory prefix where the platform independent Python files are installed ; on Unix, the default is `/usr/local`. This can be set at build time with the `--prefix` argument to the **`configure`** script. See *Installation paths* for derived paths.

> **Note**
>
> If a *virtual environment* is in effect, this `prefix` will point to the virtual environment. The value for the Python installation will still be available, via `base_prefix`. Refer to *Virtual Environments* for more information.

Modifié dans la version 3.14 : When running under a *virtual environment*, `prefix` and `exec_prefix` are now set to the virtual environment prefix by the *path initialization*, instead of `site`. This means that `prefix` and `exec_prefix` always point to the virtual environment, even when `site` is disabled (`-S`).

`sys.`**`ps1`**

`sys.`**`ps2`**

Chaînes spécifiant l'invite primaire et secondaire de l'interpréteur. Celles-ci ne sont définies que si l'interpréteur est en mode interactif. Dans ce cas, leurs valeurs initiales sont `'>>>'` et `'...'`. Si un objet qui n'est pas une chaîne est assigné à l'une ou l'autre variable, sa méthode `str()` sera appelée à chaque fois que l'interpréteur se prépare à lire une nouvelle commande interactive, c'est donc utilisable pour implémenter une invite dynamique.

`sys.`**`setdlopenflags`**(*n*)

Set the flags used by the interpreter for `dlopen()` calls, such as when the interpreter loads extension modules. Among other things, this will enable a lazy resolving of symbols when importing a module, if called as `sys.setdlopenflags(0)`. To share symbols across extension modules, call as `sys.setdlopenflags(os.RTLD_GLOBAL)`. Symbolic names for the flag values can be found in the `os` module (`RTLD_xxx` constants, e.g. `os.RTLD_LAZY`).

Availability : Unix.

`sys.`**`set_int_max_str_digits`**(*maxdigits*)

Set the *integer string conversion length limitation* used by this interpreter. See also `get_int_max_str_digits()`.

Ajouté dans la version 3.11.

`sys.`**`setprofile`**(*profilefunc*)

Définit la fonction de profilage du système, qui vous permet d'implémenter un profileur de code source Python en Python. Voir le chapitre *The Python Profilers* pour plus d'informations sur le profileur Python. La fonction de profilage du système est appelée de la même façon que la fonction trace du système (voir `settrace()`), mais elle est appelée pour des évènements différents, par exemple elle n'est pas appelée à chaque ligne de code exécutée (seulement sur appel et retours, mais l'événement pour les retours est appelé même en cas d'exception). Cette fonction est locale au fil d'exécution, et il n'existe aucun moyen, du point de vue du profileur, de prendre conscience des changements de contextes entre fils d'exécution, ça n'a donc aucun sens d'utiliser cette fonction dans un contexte *multithread*. Sa valeur de retour n'est pas utilisée, elle peut simplement renvoyer `None`.

> **Note**
>
> The same tracing mechanism is used for `setprofile()` as `settrace()`. To trace calls with `setprofile()` inside a tracing function (e.g. in a debugger breakpoint), see `call_tracing()`.

Les fonctions de traçage doivent avoir trois arguments : *frame*, *event*, et *arg*. *frame* est la *stack frame* actuelle. *event* est une chaîne de caractères pouvant valoir : `'call'`, `'return'`, `'c_call'`, `'c_return'` ou `'c_exception'`. *arg* dépend du type de l'évènement.

Les événements ont la signification suivante :

`'call'`
Une fonction est appelée (ou Python entre dans un autre bloc de code). La fonction de traçage est appelée, *arg* est `None`.

`'return'`
La fonction (ou un autre type de bloc) est sur le point de se terminer. La fonction de traçage est appelée, *arg* est la valeur qui sera renvoyée, ou `None` si l'événement est causé par la levée d'une exception.

'c_call'
Une fonction C est sur le point d'être appelée. C'est soit une fonction d'extension ou une fonction native. *arg* représente la fonction C.

'c_return'
Une fonction C a renvoyé une valeur. *arg* représente la fonction C.

'c_exception'
Une fonction C a levé une exception. *arg* représente la fonction C.

Lève un *évènement d'audit* `sys.setprofile` sans arguments.

`sys.`**`setrecursionlimit`**(*limit*)

Définit la profondeur maximale de la pile de l'interpréteur Python à *limit*. Cette limite empêche une récursion infinie de provoquer un débordement de la pile C et ainsi un crash de Python.

La limite haute dépend de la plate-forme. Un utilisateur pourrait avoir besoin de remonter la limite, lorsque son programme nécessite une récursion profonde, si sa plate-forme le permet. Cela doit être fait avec précaution, car une limite trop élevée peut conduire à un crash.

Si la nouvelle limite est plus basse que la profondeur actuelle, une `RecursionError` est levée.

Modifié dans la version 3.5.1 : Une `RecursionError` est maintenant levée si la nouvelle limite est plus basse que la profondeur de récursion actuelle.

`sys.`**`setswitchinterval`**(*interval*)

Configure l'intervalle de bascule des fils d'exécution de l'interpréteur (en secondes). Ce nombre à virgule flottante détermine la durée idéale allouée aux fils d'exécution en cour d'exécution (durée appelée *timeslices*). Notez que la durée observée peut être plus grande, typiquement si des fonctions ou méthodes prenant beaucoup de temps sont utilisées. Aussi, le choix du fil d'exécution prenant la main à la fin de l'intervalle revient au système d'exploitation. L'interpréteur n'a pas son propre ordonnanceur.

Ajouté dans la version 3.2.

`sys.`**`settrace`**(*tracefunc*)

Définit la fonction de traçage du système, qui vous permet d'implémenter un débogueur de code source Python en Python. Cette fonction est locale au fil d'exécution courant. Pour qu'un débogueur puisse gérer plusieurs fils d'exécution, il doit enregistrer sa fonction en appelant `settrace()` pour chaque fil d'exécution qu'il souhaite surveiller ou utilisez `threading.settrace()`.

Les fonctions de traçage doivent avoir trois arguments : *frame*, *event*, et *arg*. *frame* est la *stack frame* actuelle. *event* est une chaîne de caractères pouvant valoir : `'call'`, `'line'`, `'return'`, `'exception'` ou `'opcode'`. *arg* dépend du type de l'évènement.

La fonction de traçage est appelée (avec *event* à `'call'`) à chaque fois que l'interpréteur entre dans un nouveau *scope*. Elle doit renvoyer une référence à une fonction de traçage locale à utiliser pour ce *scope*, ou `None` si le *Scope* ne doit pas être tracé.

The local trace function should return a reference to itself, or to another function which would then be used as the local trace function for the scope.

Si une erreur se produit dans la fonction de trace, elle sera désactivée, tout comme si `settrace(None)` avait été appelée.

> **Note**
>
> Tracing is disabled while calling the trace function (e.g. a function set by `settrace()`). For recursive tracing see `call_tracing()`.

Les événements ont la signification suivante :

'call'
Une fonction est appelée (un un bloc de code). La fonction de traçage globale est appelée, *arg* est `None`, la valeur renvoyée donne la fonction de traçage locale.

'line'
The interpreter is about to execute a new line of code or re-execute the condition of a loop. The local trace

function is called ; *arg* is `None` ; the return value specifies the new local trace function. See `Objects/lnotab_notes.txt` for a detailed explanation of how this works. Per-line events may be disabled for a frame by setting `f_trace_lines` to `False` on that frame.

'return'
La fonction (ou un autre type de bloc) est sur le point de se terminer. La fonction de traçage locale est appelée, *arg* est la valeur qui sera renvoyée, ou `None` si l'événement est causé par la levée d'une exception. La valeur renvoyée par la fonction de traçage est ignorée.

'exception'
Une exception est survenue. La fonction de traçage locale est appelée, *arg* est le triplet `(exception, valeur, traceback)`, la valeur renvoyée spécifie la nouvelle fonction de traçage locale.

'opcode'
The interpreter is about to execute a new opcode (see `dis` for opcode details). The local trace function is called ; *arg* is `None` ; the return value specifies the new local trace function. Per-opcode events are not emitted by default : they must be explicitly requested by setting `f_trace_opcodes` to `True` on the frame.

Remarquez que, comme une exception se propage au travers de toute chaîne d'appelants, un événement `'exception'` est généré à chaque niveau.

Pour une utilisation plus fine, il est possible de définir une fonction de traçage en assignant `frame.f_trace = tracefunc` explicitement, plutôt que de s'appuyer sur le fait qu'elle soit définie indirectement par la valeur de retour d'une fonction trace déjà installée. Cela est aussi demandé pour l'activation de la fonction de traçage dans le cadre, ce que `settrace()` ne fait pas. Notez que pour que cela fonctionne, une fonction globale de traçage doit avoir été installée avec `settrace()` afin d'activer le mécanisme de traçage au moment de l'exécution, mais il n'est pas nécessaire que ce soit la même fonction de traçage (Par exemple, cela pourrait être une fonction de traçage avec peu de surcharge qui retourne simplement `None` pour se désactiver immédiatement à chaque cadre).

Pour plus d'informations sur les objets code et objets représentant une *frame* de la pile, consultez types.

Lève un *évènement d'audit* `sys.settrace` sans arguments.

La fonction `settrace()` est destinée uniquement à l'implémentation de débogueurs, de profileurs, d'outils d'analyse de couverture et d'autres outils similaires. Son comportement fait partie de l'implémentation, plutôt que de la définition du langage, et peut donc ne pas être disponible dans toutes les implémentations de Python.

Modifié dans la version 3.7 : `'opcode'` event type added ; `f_trace_lines` and `f_trace_opcodes` attributes added to frames

`sys.`**`set_asyncgen_hooks`**(*[firstiter] [, finalizer]*)

Accepte deux arguments optionnels nommés, qui sont appelables qui acceptent un *asynchronous generator iterator* comme argument. L'appelable *firsttiter* sera appelé lorsqu'un générateur asynchrone sera itéré pour la première fois, et l'appelable *finalizer* sera appelé lorsqu'un générateur asynchrone est sur le point d'être détruit.

Lève un *évènement d'audit* `sys.set_asyncgen_hooks_firstiter` sans arguments.

Lève un *événement d'audit* `sys.set_asyncgen_hooks_finalizer` sans arguments.

Deux événements d'audit sont levés car l'API sous-jacente consiste de deux appels, dont chacun doit lever son propre événement.

Ajouté dans la version 3.6 : Voir la **PEP 525** pour plus de détails. Pour un exemple de *finalizer*, voir l'implémentation de `asyncio.Loop.shutdown_asyncgens` dans Lib/asyncio/base_events.py

> **Note**
>
> Cette fonction à été ajoutée à titre provisoire (voir la **PEP 411** pour plus d'informations.)

`sys.`**`set_coroutine_origin_tracking_depth`**(*depth*)

Allows enabling or disabling coroutine origin tracking. When enabled, the `cr_origin` attribute on coroutine objects will contain a tuple of (filename, line number, function name) tuples describing the traceback where the coroutine object was created, with the most recent call first. When disabled, `cr_origin` will be `None`.

To enable, pass a *depth* value greater than zero ; this sets the number of frames whose information will be

captured. To disable, set *depth* to zero.

Ce paramètre est spécifique au fil d'exécution courant.

Ajouté dans la version 3.7.

> **Note**
>
> Cette fonction à été ajoutée à titre provisoire (Voir la **PEP 411** pour plus d'informations.) Utilisez la uniquement à des fins de débogage.

`sys.activate_stack_trampoline`(*backend*, /)

Activate the stack profiler trampoline *backend*. The only supported backend is `"perf"`.

Stack trampolines cannot be activated if the JIT is active.

Availability : Linux.

Ajouté dans la version 3.12.

> **Voir aussi**
>
> — perf_profiling
> — https://perf.wiki.kernel.org

`sys.deactivate_stack_trampoline`()

Deactivate the current stack profiler trampoline backend.

If no stack profiler is activated, this function has no effect.

Availability : Linux.

Ajouté dans la version 3.12.

`sys.is_stack_trampoline_active`()

Return `True` if a stack profiler trampoline is active.

Availability : Linux.

Ajouté dans la version 3.12.

`sys._enablelegacywindowsfsencoding`()

Changes the *filesystem encoding and error handler* to 'mbcs' and 'replace' respectively, for consistency with versions of Python prior to 3.6.

Équivaut à définir la variable d'environnement `PYTHONLEGACYWINDOWSFSENCODING` avant de lancer Python.

See also `sys.getfilesystemencoding()` and `sys.getfilesystemencodeerrors()`.

Availability : Windows.

> **Note**
>
> Changing the filesystem encoding after Python startup is risky because the old fsencoding or paths encoded by the old fsencoding may be cached somewhere. Use `PYTHONLEGACYWINDOWSFSENCODING` instead.

Ajouté dans la version 3.6 : Voir la **PEP 529** pour plus d'informations.

Deprecated since version 3.13, will be removed in version 3.16 : Use `PYTHONLEGACYWINDOWSFSENCODING` instead.

`sys.stdin`

`sys.stdout`

sys.**stderr**

objets fichiers utilisés par l'interpréteur pour l'entrée standard, la sortie standard et la sortie d'erreurs :

— `stdin` est utilisé pour toutes les entrées interactives (y compris les appels à `input()`)
— `stdout` est utilisé pour la sortie de `print()`, des *expression* et pour les invites de `input()` ;
— Les invites de l'interpréteur et ses messages d'erreur sont écrits sur `stderr`.

Ces flux sont de classiques *fichiers texte* comme ceux renvoyés par la fonction `open()`. Leurs paramètres sont choisis comme suit :

— The encoding and error handling are is initialized from `PyConfig.stdio_encoding` and `PyConfig.stdio_errors`.

  On Windows, UTF-8 is used for the console device. Non-character devices such as disk files and pipes use the system locale encoding (i.e. the ANSI codepage). Non-console character devices such as NUL (i.e. where `isatty()` returns `True`) use the value of the console input and output codepages at startup, respectively for stdin and stdout/stderr. This defaults to the system *locale encoding* if the process is not initially attached to a console.

  Le comportement spécial de la console peut être redéfini en assignant la variable d'environnement *PYTHONLEGACYWINDOWSSTDIO* avant de démarrer Python. Dans ce cas, les pages de code de la console sont utilisées comme pour tout autre périphérique de caractères.

  Sous toutes les plateformes, vous pouvez redéfinir le codage de caractères en assignant la variable d'environnement `PYTHONIOENCODING` avant de démarrer Python ou en utilisant la nouvelle option de ligne de commande `-X utf8` et la variable d'environnement `PYTHONUTF8`. Toutefois, pour la console Windows, cela s'applique uniquement lorsque `PYTHONLEGACYWINDOWSSTDIO` est également défini.
— When interactive, the `stdout` stream is line-buffered. Otherwise, it is block-buffered like regular text files. The `stderr` stream is line-buffered in both cases. You can make both streams unbuffered by passing the `-u` command-line option or setting the `PYTHONUNBUFFERED` environment variable.

Modifié dans la version 3.9 : Le `stderr` non interactif est maintenant mis en mémoire-tampon ligne par ligne plutôt qu'entièrement.

> **Note**
>
> Pour écrire ou lire des données binaires depuis ou vers les flux standards, utilisez l'objet sous-jacent `buffer`. Par exemple, pour écrire des octets sur `stdout`, utilisez `sys.stdout.buffer.write(b'abc')`.
>
> However, if you are writing a library (and do not control in which context its code will be executed), be aware that the standard streams may be replaced with file-like objects like `io.StringIO` which do not support the `buffer` attribute.

sys.**__stdin__**

sys.**__stdout__**

sys.**__stderr__**

Ces objets contiennent les valeurs d'origine de `stdin`, `stderr` et `stdout` tel que présentes au début du programme. Ils sont utilisés pendant la finalisation, et peuvent être utiles pour écrire dans le vrai flux standard, peu importe si l'objet `sys.std*` a été redirigé.

Ils peuvent également être utilisés pour restaurer les entrées / sorties d'origine, au cas où ils auraient été écrasés par des objets cassé, cependant la bonne façon de faire serait de sauvegarder explicitement les flux avant de les remplacer et ainsi pouvoir les restaurer.

> **Note**
>
> Dans certaines cas, `stdin`, `stdout` et `stderr` ainsi que les valeurs initiales `__stdin__`, `__stdout__` et `__stderr__` peuvent être `None`. C'est typiquement le cas pour les applications graphiques sur Windows qui ne sont pas connectées à une console, ou les applications Python démarrées avec **pythonw**.

sys.**stdlib_module_names**

A frozenset of strings containing the names of standard library modules.

It is the same on all platforms. Modules which are not available on some platforms and modules disabled at Python build are also listed. All module kinds are listed : pure Python, built-in, frozen and extension modules. Test modules are excluded.

For packages, only the main package is listed : sub-packages and sub-modules are not listed. For example, the `email` package is listed, but the `email.mime` sub-package and the `email.message` sub-module are not listed.

See also the `sys.builtin_module_names` list.

Ajouté dans la version 3.10.

`sys.`**`thread_info`**

Un *named tuple* contenant des informations sur l'implémentation des fils d'exécution.

`thread_info.`**`name`**

The name of the thread implementation :
— `"nt"` : Windows threads
— `"pthread"` : POSIX threads
— `"pthread-stubs"` : stub POSIX threads (on WebAssembly platforms without threading support)
— `"solaris"` : Solaris threads

`thread_info.`**`lock`**

The name of the lock implementation :
— `"semaphore"` : a lock uses a semaphore
— `"mutex+cond"` : a lock uses a mutex and a condition variable
— `None` si cette information n'est pas connue

`thread_info.`**`version`**

The name and version of the thread library. It is a string, or `None` if this information is unknown.

Ajouté dans la version 3.3.

`sys.`**`tracebacklimit`**

Lorsque cette variable contient un nombre entier, elle détermine la profondeur maximum de la pile d'appels affichée lorsqu'une exception non gérée se produit. La valeur par défaut est `1000`, lorsque cette valeur est égale ou inférieure à `0`, la pile d'appels n'est pas affichée, seul seuls le type et la valeur de l'exception sont le sont.

`sys.`**`unraisablehook`**(*unraisable*, /)

Gère une exception *non levable* (**unraisable** en anglais)

Appelé lorsqu'une exception s'est produite mais qu'il n'y a aucun moyen pour Python de la gérer. Par exemple, lorsqu'un destructeur lève une exception ou durant le passage du ramasse-miettes (`gc.collect()`).

Les arguments *unraisable* ont la signification suivante :
— `exc_type` : Exception type.
— `exc_value` : Exception value, can be `None`.
— `exc_traceback` : Exception traceback, can be `None`.
— `err_msg` : Error message, can be `None`.
— `object` : Object causing the exception, can be `None`.
The default hook formats `err_msg` and `object` as : `f'{err_msg}: {object!r}'` ; use "Exception ignored in" error message if `err_msg` is `None`.

`sys.unraisablehook()` peut être remplacé pour contrôler comment les exceptions *non levables* (**unraisable** en anglais) sont gérées.

Voir aussi

`excepthook()` which handles uncaught exceptions.

> **Avertissement**
>
> Storing `exc_value` using a custom hook can create a reference cycle. It should be cleared explicitly to break the reference cycle when the exception is no longer needed.
>
> Storing `object` using a custom hook can resurrect it if it is set to an object which is being finalized. Avoid storing `object` after the custom hook completes to avoid resurrecting objects.

Raise an auditing event `sys.unraisablehook` with arguments *hook*, *unraisable* when an exception that cannot be handled occurs. The *unraisable* object is the same as what will be passed to the hook. If no hook has been set, *hook* may be `None`.

Ajouté dans la version 3.8.

`sys.`**`version`**

Une chaîne contenant le numéro de version de l'interpréteur Python, ainsi que d'autres informations comme le numéro de compilation et le compilateur utilisé. Cette chaîne est affichée lorsque l'interpréteur est démarré en mode interactif. N'essayez pas d'en extraire des informations de version, utilisez plutôt `version_info` et les fonctions fournies par le module `platform`.

`sys.`**`api_version`**

La version de l'API C pour cet interpréteur. Les développeurs peuvent trouver cette information utile en déboguant des conflits de versions entre Python et des modules d'extension.

`sys.`**`version_info`**

Un quintuplet contenant les composants du numéro de version : *major*, *minor*, *micro*, *releaselevel* et *serial*. Toutes les valeurs sauf *releaselevel* sont des nombres entiers. *releaselevel* peut valoir `'alpha'`, `'beta'`, `'candidate'`, ou `'final'`. La valeur de `version_info` pour Python 2.0 est `(2, 0, 0, 'final', 0)`. Ces attributs sont aussi accessibles par leur nom, ainsi `sys.version_info[0]` est équivalent à `sys.version_info.major`, et ainsi de suite.

Modifié dans la version 3.1 : Ajout des attributs nommés.

`sys.`**`warnoptions`**

C'est une spécificité de l'implémentation de la gestion des avertissements. Ne modifiez pas cette valeur. Reportez-vous au module `warnings` pour plus d'informations sur le gestionnaire d'avertissements.

`sys.`**`winver`**

The version number used to form registry keys on Windows platforms. This is stored as string resource 1000 in the Python DLL. The value is normally the major and minor versions of the running Python interpreter. It is provided in the `sys` module for informational purposes; modifying this value has no effect on the registry keys used by Python.

Availability : Windows.

`sys.`**`monitoring`**

Namespace containing functions and constants for register callbacks and controlling monitoring events. See `sys.monitoring` for details.

`sys.`**`_xoptions`**

Un dictionnaire des différentes options spécifiques à l'implémentation passés en ligne de commande via l'option `-X`. Aux noms des options correspondent soit leur valeur, si elle est donnée explicitement, soit à `True`. Exemple :

```
$ ./python -Xa=b -Xc
Python 3.2a3+ (py3k, Oct 16 2010, 20:14:50)
[GCC 4.4.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> sys._xoptions
{'a': 'b', 'c': True}
```

C'est un moyen spécifique à CPython pour accéder aux options passées via l'option `-X`. D'autres implémentations pourraient les exposer par d'autres moyens, ou pas du tout.

Ajouté dans la version 3.2.

Citations

30.2 `sys.monitoring` --- Execution event monitoring

Ajouté dans la version 3.12.

> **Note**
>
> `sys.monitoring` is a namespace within the `sys` module, not an independent module, so there is no need to `import sys.monitoring`, simply `import sys` and then use `sys.monitoring`.

This namespace provides access to the functions and constants necessary to activate and control event monitoring.

As programs execute, events occur that might be of interest to tools that monitor execution. The `sys.monitoring` namespace provides means to receive callbacks when events of interest occur.

The monitoring API consists of three components :

- *Tool identifiers*
- *Events*
- *Callbacks*

30.2.1 Tool identifiers

A tool identifier is an integer and the associated name. Tool identifiers are used to discourage tools from interfering with each other and to allow multiple tools to operate at the same time. Currently tools are completely independent and cannot be used to monitor each other. This restriction may be lifted in the future.

Before registering or activating events, a tool should choose an identifier. Identifiers are integers in the range 0 to 5 inclusive.

Registering and using tools

`sys.monitoring.`**`use_tool_id`**(*tool_id* : int, *name* : str, /) → *None*

Must be called before *tool_id* can be used. *tool_id* must be in the range 0 to 5 inclusive. Raises a `ValueError` if *tool_id* is in use.

`sys.monitoring.`**`clear_tool_id`**(*tool_id* : int, /) → *None*

Unregister all events and callback functions associated with *tool_id*.

`sys.monitoring.`**`free_tool_id`**(*tool_id* : int, /) → *None*

Should be called once a tool no longer requires *tool_id*. Will call `clear_tool_id()` before releasing *tool_id*.

`sys.monitoring.`**`get_tool`**(*tool_id* : int, /) → *str* | *None*

Returns the name of the tool if *tool_id* is in use, otherwise it returns `None`. *tool_id* must be in the range 0 to 5 inclusive.

All IDs are treated the same by the VM with regard to events, but the following IDs are pre-defined to make co-operation of tools easier :

```
sys.monitoring.DEBUGGER_ID = 0
sys.monitoring.COVERAGE_ID = 1
sys.monitoring.PROFILER_ID = 2
sys.monitoring.OPTIMIZER_ID = 5
```

30.2.2 Events

The following events are supported :

`sys.monitoring.events.`**BRANCH_LEFT**

A conditional branch goes left.

It is up to the tool to determine how to present "left" and "right" branches. There is no guarantee which branch is "left" and which is "right", except that it will be consistent for the duration of the program.

`sys.monitoring.events.`**BRANCH_RIGHT**

A conditional branch goes right.

`sys.monitoring.events.`**CALL**

A call in Python code (event occurs before the call).

`sys.monitoring.events.`**C_RAISE**

An exception raised from any callable, except for Python functions (event occurs after the exit).

`sys.monitoring.events.`**C_RETURN**

Return from any callable, except for Python functions (event occurs after the return).

`sys.monitoring.events.`**EXCEPTION_HANDLED**

An exception is handled.

`sys.monitoring.events.`**INSTRUCTION**

A VM instruction is about to be executed.

`sys.monitoring.events.`**JUMP**

An unconditional jump in the control flow graph is made.

`sys.monitoring.events.`**LINE**

An instruction is about to be executed that has a different line number from the preceding instruction.

`sys.monitoring.events.`**PY_RESUME**

Resumption of a Python function (for generator and coroutine functions), except for `throw()` calls.

`sys.monitoring.events.`**PY_RETURN**

Return from a Python function (occurs immediately before the return, the callee's frame will be on the stack).

`sys.monitoring.events.`**PY_START**

Start of a Python function (occurs immediately after the call, the callee's frame will be on the stack)

`sys.monitoring.events.`**PY_THROW**

A Python function is resumed by a `throw()` call.

`sys.monitoring.events.`**PY_UNWIND**

Exit from a Python function during exception unwinding.

`sys.monitoring.events.`**PY_YIELD**

Yield from a Python function (occurs immediately before the yield, the callee's frame will be on the stack).

`sys.monitoring.events.`**RAISE**

An exception is raised, except those that cause a *`STOP_ITERATION`* event.

`sys.monitoring.events.`**RERAISE**

An exception is re-raised, for example at the end of a `finally` block.

`sys.monitoring.events.`**`STOP_ITERATION`**

An artificial `StopIteration` is raised; see *the STOP_ITERATION event*.

More events may be added in the future.

These events are attributes of the `sys.monitoring.events` namespace. Each event is represented as a power-of-2 integer constant. To define a set of events, simply bitwise or the individual events together. For example, to specify both `PY_RETURN` and `PY_START` events, use the expression `PY_RETURN | PY_START`.

`sys.monitoring.events.`**`NO_EVENTS`**

An alias for `0` so users can do explicit comparisons like :

```
if get_events(DEBUGGER_ID) == NO_EVENTS:
    ...
```

Events are divided into three groups :

Local events

Local events are associated with normal execution of the program and happen at clearly defined locations. All local events can be disabled. The local events are :

- `PY_START`
- `PY_RESUME`
- `PY_RETURN`
- `PY_YIELD`
- `CALL`
- `LINE`
- `INSTRUCTION`
- `JUMP`
- `BRANCH_LEFT`
- `BRANCH_RIGHT`
- `STOP_ITERATION`

Deprecated event

- `BRANCH`

The `BRANCH` event is deprecated in 3.14. Using `BRANCH_LEFT` and `BRANCH_RIGHT` events will give much better performance as they can be disabled independently.

Ancillary events

Ancillary events can be monitored like other events, but are controlled by another event :

- `C_RAISE`
- `C_RETURN`

The `C_RETURN` and `C_RAISE` events are controlled by the `CALL` event. `C_RETURN` and `C_RAISE` events will only be seen if the corresponding `CALL` event is being monitored.

Other events

Other events are not necessarily tied to a specific location in the program and cannot be individually disabled.

The other events that can be monitored are :

- `PY_THROW`
- `PY_UNWIND`
- `RAISE`
- `EXCEPTION_HANDLED`

The STOP_ITERATION event

PEP 380 specifies that a `StopIteration` exception is raised when returning a value from a generator or coroutine. However, this is a very inefficient way to return a value, so some Python implementations, notably CPython 3.12+, do not raise an exception unless it would be visible to other code.

To allow tools to monitor for real exceptions without slowing down generators and coroutines, the `STOP_ITERATION` event is provided. `STOP_ITERATION` can be locally disabled, unlike `RAISE`.

Note that the `STOP_ITERATION` event and the `RAISE` event for a `StopIteration` exception are equivalent, and are treated as interchangeable when generating events. Implementations will favor `STOP_ITERATION` for performance reasons, but may generate a `RAISE` event with a `StopIteration`.

30.2.3 Turning events on and off

In order to monitor an event, it must be turned on and a corresponding callback must be registered. Events can be turned on or off by setting the events either globally or for a particular code object.

Setting events globally

Events can be controlled globally by modifying the set of events being monitored.

`sys.monitoring.`**`get_events`**`(`*tool_id* : int, `/)` → *int*

Returns the `int` representing all the active events.

`sys.monitoring.`**`set_events`**`(`*tool_id* : int, *event_set* : int, `/)` → *None*

Activates all events which are set in *event_set*. Raises a `ValueError` if *tool_id* is not in use.

No events are active by default.

Per code object events

Events can also be controlled on a per code object basis. The functions defined below which accept a `types.CodeType` should be prepared to accept a look-alike object from functions which are not defined in Python (see c-api-monitoring).

`sys.monitoring.`**`get_local_events`**`(`*tool_id* : int, *code* : CodeType, `/)` → *int*

Returns all the local events for *code*

`sys.monitoring.`**`set_local_events`**`(`*tool_id* : int, *code* : CodeType, *event_set* : int, `/)` → *None*

Activates all the local events for *code* which are set in *event_set*. Raises a `ValueError` if *tool_id* is not in use.

Local events add to global events, but do not mask them. In other words, all global events will trigger for a code object, regardless of the local events.

Disabling events

`sys.monitoring.`**`DISABLE`**

A special value that can be returned from a callback function to disable events for the current code location.

Local events can be disabled for a specific code location by returning `sys.monitoring.DISABLE` from a callback function. This does not change which events are set, or any other code locations for the same event.

Disabling events for specific locations is very important for high performance monitoring. For example, a program can be run under a debugger with no overhead if the debugger disables all monitoring except for a few breakpoints.

`sys.monitoring.`**`restart_events`**`()` → *None*

Enable all the events that were disabled by `sys.monitoring.DISABLE` for all tools.

30.2.4 Registering callback functions

To register a callable for events call

`sys.monitoring.register_callback`(*tool_id* : int, *event* : int, *func* : Callable | None, /) → *Callable* | *None*

Registers the callable *func* for the *event* with the given *tool_id*

If another callback was registered for the given *tool_id* and *event*, it is unregistered and returned. Otherwise `register_callback()` returns `None`.

Functions can be unregistered by calling `sys.monitoring.register_callback(tool_id, event, None)`.

Callback functions can be registered and unregistered at any time.

Registering or unregistering a callback function will generate a `sys.audit()` event.

Callback function arguments

`sys.monitoring.MISSING`

A special value that is passed to a callback function to indicate that there are no arguments to the call.

When an active event occurs, the registered callback function is called. Different events will provide the callback function with different arguments, as follows :

— `PY_START` and `PY_RESUME` :

```
func(code: CodeType, instruction_offset: int) -> DISABLE | Any
```

— `PY_RETURN` and `PY_YIELD` :

```
func(code: CodeType, instruction_offset: int, retval: object) -> DISABLE | Any
```

— `CALL`, `C_RAISE` and `C_RETURN` :

```
func(code: CodeType, instruction_offset: int, callable: object, arg0: object␣
↪| MISSING) -> DISABLE | Any
```

If there are no arguments, *arg0* is set to `sys.monitoring.MISSING`.

— `RAISE`, `RERAISE`, `EXCEPTION_HANDLED`, `PY_UNWIND`, `PY_THROW` and `STOP_ITERATION` :

```
func(code: CodeType, instruction_offset: int, exception: BaseException) ->␣
↪DISABLE | Any
```

— `LINE` :

```
func(code: CodeType, line_number: int) -> DISABLE | Any
```

— `BRANCH_LEFT`, `BRANCH_RIGHT` and `JUMP` :

```
func(code: CodeType, instruction_offset: int, destination_offset: int) ->␣
↪DISABLE | Any
```

Note that the *destination_offset* is where the code will next execute.

— `INSTRUCTION` :

```
func(code: CodeType, instruction_offset: int) -> DISABLE | Any
```

30.3 `sysconfig` --- Provide access to Python's configuration information

Ajouté dans la version 3.2.

Source code : Lib/sysconfig

The *sysconfig* module provides access to Python's configuration information like the list of installation paths and the configuration variables relevant for the current platform.

30.3.1 Configuration variables

A Python distribution contains a `Makefile` and a `pyconfig.h` header file that are necessary to build both the Python binary itself and third-party C extensions compiled using `setuptools`.

sysconfig puts all variables found in these files in a dictionary that can be accessed using *get_config_vars()* or *get_config_var()*.

Notice that on Windows, it's a much smaller set.

`sysconfig.`**`get_config_vars`**(**args*)

With no arguments, return a dictionary of all configuration variables relevant for the current platform.

With arguments, return a list of values that result from looking up each argument in the configuration variable dictionary.

For each argument, if the value is not found, return `None`.

`sysconfig.`**`get_config_var`**(*name*)

Return the value of a single variable *name*. Equivalent to `get_config_vars().get(name)`.

If *name* is not found, return `None`.

Example of usage :

```
>>> import sysconfig
>>> sysconfig.get_config_var('Py_ENABLE_SHARED')
0
>>> sysconfig.get_config_var('LIBDIR')
'/usr/local/lib'
>>> sysconfig.get_config_vars('AR', 'CXX')
['ar', 'g++']
```

30.3.2 Installation paths

Python uses an installation scheme that differs depending on the platform and on the installation options. These schemes are stored in *sysconfig* under unique identifiers based on the value returned by *os.name*. The schemes are used by package installers to determine where to copy files to.

Python currently supports nine schemes :

- *posix_prefix* : scheme for POSIX platforms like Linux or macOS. This is the default scheme used when Python or a component is installed.
- *posix_home* : scheme for POSIX platforms, when the *home* option is used. This scheme defines paths located under a specific home prefix.
- *posix_user* : scheme for POSIX platforms, when the *user* option is used. This scheme defines paths located under the user's home directory (*site.USER_BASE*).
- *posix_venv* : scheme for *Python virtual environments* on POSIX platforms ; by default it is the same as *posix_prefix*.
- *nt* : scheme for Windows. This is the default scheme used when Python or a component is installed.
- *nt_user* : scheme for Windows, when the *user* option is used.
- *nt_venv* : scheme for *Python virtual environments* on Windows ; by default it is the same as *nt*.
- *venv* : a scheme with values from either *posix_venv* or *nt_venv* depending on the platform Python runs on.
- *osx_framework_user* : scheme for macOS, when the *user* option is used.

Each scheme is itself composed of a series of paths and each path has a unique identifier. Python currently uses eight paths :

- *stdlib* : directory containing the standard Python library files that are not platform-specific.
- *platstdlib* : directory containing the standard Python library files that are platform-specific.
- *platlib* : directory for site-specific, platform-specific files.
- *purelib* : directory for site-specific, non-platform-specific files ('pure' Python).

— *include* : directory for non-platform-specific header files for the Python C-API.
— *platinclude* : directory for platform-specific header files for the Python C-API.
— *scripts* : directory for script files.
— *data* : directory for data files.

30.3.3 User scheme

This scheme is designed to be the most convenient solution for users that don't have write permission to the global site-packages directory or don't want to install into it.

Files will be installed into subdirectories of `site.USER_BASE` (written as `userbase` hereafter). This scheme installs pure Python modules and extension modules in the same location (also known as `site.USER_SITE`).

posix_user

Path	Dossier d'installation
stdlib	`userbase/lib/pythonX.Y`
platstdlib	`userbase/lib/pythonX.Y`
platlib	`userbase/lib/pythonX.Y/site-packages`
purelib	`userbase/lib/pythonX.Y/site-packages`
include	`userbase/include/pythonX.Y`
scripts	`userbase/bin`
données	`userbase`

nt_user

Path	Dossier d'installation
stdlib	`userbase\PythonXY`
platstdlib	`userbase\PythonXY`
platlib	`userbase\PythonXY\site-packages`
purelib	`userbase\PythonXY\site-packages`
include	`userbase\PythonXY\Include`
scripts	`userbase\PythonXY\Scripts`
données	`userbase`

osx_framework_user

Path	Dossier d'installation
stdlib	`userbase/lib/python`
platstdlib	`userbase/lib/python`
platlib	`userbase/lib/python/site-packages`
purelib	`userbase/lib/python/site-packages`
include	`userbase/include/pythonX.Y`
scripts	`userbase/bin`
données	`userbase`

30.3.4 Home scheme

L'idée derrière le « schéma home » est que vous compilez et maintenez un espace personnel de modules Python. Le nom de ce schéma vient de l'idée du dossier « home » sur Unix, vu qu'il n'est pas rare pour un utilisateur UNIX d'agencer son dossier *home* avec la même disposition que `/usr/` ou `/usr/local/`. Ce schéma peut être utilisé par n'importe qui, quel que soit le système d'exploitation.

posix_home

Path	Dossier d'installation
stdlib	*home*/lib/python
platstdlib	*home*/lib/python
platlib	*home*/lib/python
purelib	*home*/lib/python
include	*home*/include/python
platinclude	*home*/include/python
scripts	*home*/bin
données	*home*

30.3.5 Prefix scheme

Le schéma de préfixe est utile quand vous voulez une installation de Python pour faire la compilation/l'installation (c.-à-d. exécuter le script *setup*), mais utiliser les modules tiers d'une installation Python différente (ou quelque chose qui ressemble à une installation Python différente). Si cela vous semble inhabituel, ça l'est — c'est pourquoi les schémas *user* et *home* viennent avant. Cependant, il y a au moins deux cas connus où le schéma *prefix* est utile.

First, consider that many Linux distributions put Python in `/usr`, rather than the more traditional `/usr/local`. This is entirely appropriate, since in those cases Python is part of "the system" rather than a local add-on. However, if you are installing Python modules from source, you probably want them to go in `/usr/local/lib/python2.`*X* rather than `/usr/lib/python2.`*X*.

Another possibility is a network filesystem where the name used to write to a remote directory is different from the name used to read it : for example, the Python interpreter accessed as `/usr/local/bin/python` might search for modules in `/usr/local/lib/python2.`*X*, but those modules would have to be installed to, say, `/mnt/`*@server*`/export/lib/python2.`*X*.

posix_prefix

Path	Dossier d'installation
stdlib	*prefix*/lib/python*X.Y*
platstdlib	*prefix*/lib/python*X.Y*
platlib	*prefix*/lib/python*X.Y*/site-packages
purelib	*prefix*/lib/python*X.Y*/site-packages
include	*prefix*/include/python*X.Y*
platinclude	*prefix*/include/python*X.Y*
scripts	*prefix*/bin
données	*prefix*

nt

Path	Dossier d'installation
stdlib	*prefix*\Lib
platstdlib	*prefix*\Lib
platlib	*prefix*\Lib\site-packages
purelib	*prefix*\Lib\site-packages
include	*prefix*\Include
platinclude	*prefix*\Include
scripts	*prefix*\Scripts
données	*prefix*

30.3.6 Installation path functions

sysconfig provides some functions to determine these installation paths.

sysconfig.**get_scheme_names**()

Return a tuple containing all schemes currently supported in *sysconfig*.

sysconfig.**get_default_scheme**()

Return the default scheme name for the current platform.

Ajouté dans la version 3.10 : This function was previously named `_get_default_scheme()` and considered an implementation detail.

Modifié dans la version 3.11 : When Python runs from a virtual environment, the *venv* scheme is returned.

sysconfig.**get_preferred_scheme**(*key*)

Return a preferred scheme name for an installation layout specified by *key*.

key must be either `"prefix"`, `"home"`, or `"user"`.

The return value is a scheme name listed in *get_scheme_names()*. It can be passed to *sysconfig* functions that take a *scheme* argument, such as *get_paths()*.

Ajouté dans la version 3.10.

Modifié dans la version 3.11 : When Python runs from a virtual environment and `key="prefix"`, the *venv* scheme is returned.

sysconfig.**_get_preferred_schemes**()

Return a dict containing preferred scheme names on the current platform. Python implementers and redistributors may add their preferred schemes to the `_INSTALL_SCHEMES` module-level global value, and modify this function to return those scheme names, to e.g. provide different schemes for system and language package managers to use, so packages installed by either do not mix with those by the other.

End users should not use this function, but *get_default_scheme()* and *get_preferred_scheme()* instead.

Ajouté dans la version 3.10.

sysconfig.**get_path_names**()

Return a tuple containing all path names currently supported in *sysconfig*.

sysconfig.**get_path**(*name*[, *scheme*[, *vars*[, *expand*]]])

Return an installation path corresponding to the path *name*, from the install scheme named *scheme*.

name has to be a value from the list returned by *get_path_names()*.

sysconfig stores installation paths corresponding to each path name, for each platform, with variables to be expanded. For instance the *stdlib* path for the *nt* scheme is : `{base}/Lib`.

get_path() will use the variables returned by *get_config_vars()* to expand the path. All variables have default values for each platform so one may call this function and get the default value.

If *scheme* is provided, it must be a value from the list returned by *get_scheme_names()*. Otherwise, the default scheme for the current platform is used.

If *vars* is provided, it must be a dictionary of variables that will update the dictionary returned by *get_config_vars()*.

If *expand* is set to `False`, the path will not be expanded using the variables.

If *name* is not found, raise a *KeyError*.

sysconfig.**get_paths**([*scheme*[, *vars*[, *expand*]]])

Return a dictionary containing all installation paths corresponding to an installation scheme. See *get_path()* for more information.

If *scheme* is not provided, will use the default scheme for the current platform.

If *vars* is provided, it must be a dictionary of variables that will update the dictionary used to expand the paths.

If *expand* is set to false, the paths will not be expanded.

If *scheme* is not an existing scheme, `get_paths()` will raise a `KeyError`.

30.3.7 Autres fonctions

`sysconfig.get_python_version()`

Return the `MAJOR.MINOR` Python version number as a string. Similar to `'%d.%d' % sys.version_info[:2]`.

`sysconfig.get_platform()`

Return a string that identifies the current platform.

This is used mainly to distinguish platform-specific build directories and platform-specific built distributions. Typically includes the OS name and version and the architecture (as supplied by `os.uname()`), although the exact information included depends on the OS ; e.g., on Linux, the kernel version isn't particularly important.

Exemples de valeurs renvoyées :
— linux-i586
— linux-alpha (?)
— solaris-2.6-sun4u
Windows will return one of :
— win-amd64 (64-bit Windows on AMD64, aka x86_64, Intel64, and EM64T)
— win-arm64 (64-bit Windows on ARM64, aka AArch64)
— win32 (all others - specifically, sys.platform is returned)
macOS can return :
— macosx-10.6-ppc
— macosx-10.4-ppc64
— macosx-10.3-i386
— macosx-10.4-fat
For other non-POSIX platforms, currently just returns `sys.platform`.

`sysconfig.is_python_build()`

Return `True` if the running Python interpreter was built from source and is being run from its built location, and not from a location resulting from e.g. running `make install` or installing via a binary installer.

`sysconfig.parse_config_h(fp[, vars])`

Parse a `config.h`-style file.

fp is a file-like object pointing to the `config.h`-like file.

A dictionary containing name/value pairs is returned. If an optional dictionary is passed in as the second argument, it is used instead of a new dictionary, and updated with the values read in the file.

`sysconfig.get_config_h_filename()`

Return the path of `pyconfig.h`.

`sysconfig.get_makefile_filename()`

Return the path of `Makefile`.

30.3.8 Using `sysconfig` as a script

You can use `sysconfig` as a script with Python's *-m* option :

```
$ python -m sysconfig
Platform: "macosx-10.4-i386"
Python version: "3.2"
Current installation scheme: "posix_prefix"

Paths:
```

(suite sur la page suivante)

(suite de la page précédente)

```
        data = "/usr/local"
        include = "/Users/tarek/Dev/svn.python.org/py3k/Include"
        platinclude = "."
        platlib = "/usr/local/lib/python3.2/site-packages"
        platstdlib = "/usr/local/lib/python3.2"
        purelib = "/usr/local/lib/python3.2/site-packages"
        scripts = "/usr/local/bin"
        stdlib = "/usr/local/lib/python3.2"

Variables:
        AC_APPLE_UNIVERSAL_BUILD = "0"
        AIX_GENUINE_CPLUSPLUS = "0"
        AR = "ar"
        ARFLAGS = "rc"
        ...
```

This call will print in the standard output the information returned by *get_platform()*, *get_python_version()*, *get_path()* and *get_config_vars()*.

30.4 builtins --- Built-in objects

This module provides direct access to all 'built-in' identifiers of Python ; for example, `builtins.open` is the full name for the built-in function *open()*.

Ce module n'est normalement pas accédé explicitement par la plupart des applications, mais peut être utile dans des modules qui exposent des objets de même nom qu'une valeur native, mais pour qui le natif de même nom est aussi nécessaire. Par exemple, dans un module qui voudrait implémenter une fonction *open()* autour de la fonction native *open()*, ce module peut être utilisé directement :

```
import builtins

def open(path):
    f = builtins.open(path, 'r')
    return UpperCaser(f)

class UpperCaser:
    '''Wrapper around a file that converts output to uppercase.'''

    def __init__(self, f):
        self._f = f

    def read(self, count=-1):
        return self._f.read(count).upper()

    # ...
```

Spécificité de l'implémentation : La plupart des modules ont `__builtins__` dans leurs globales. La valeur de `__builtins__` est classiquement soit ce module, soit la valeur de l'attribut `__dict__` du module. Puisque c'est une spécificité de CPython, ce n'est peut-être pas utilisé par toutes les autres implémentations.

Voir aussi

— *Constantes natives*
— *Exceptions natives*

— *Fonctions natives*
— *Types natifs*

30.5 __main__ --- Top-level code environment

En Python, le nom `__main__` a une fonction particulière. Il intervient dans deux cas :

1. c'est le nom de l'environnement d'exécution principal, ce qui donne lieu au test courant `__name__ == '__main__'`;
2. c'est aussi le nom du fichier `__main__.py` dans les paquets Python.

Les deux sont liés aux modules Python, à la manière de s'en servir en tant qu'utilisateur et à la manière dont ils interagissent entre eux. Cette page contient des détails sur les modules Python. Si vous ne les avez jamais utilisés, commencez par la section qui leur est consacrée dans le tutoriel : tut-modules.

30.5.1 __name__ == '__main__'

Lorsqu'un module ou un paquet Python est importé, son attribut `__name__` est défini à son nom, qui est la plupart du temps le nom du fichier qui le contient sans l'extension `.py` :

```
>>> import configparser
>>> configparser.__name__
'configparser'
```

Si le fichier fait partie d'un paquet, `__name__` donne tout le chemin d'accès :

```
>>> from concurrent.futures import process
>>> process.__name__
'concurrent.futures.process'
```

En revanche, si le module est exécuté dans l'environnement d'exécution principal, la variable `__name__` vaut `'__main__'`.

Qu'est-ce que l'« environnement d'exécution principal » ?

L'environnement principal a pour nom `__main__`. Il s'agit du premier module Python dont l'exécution a été demandée par l'utilisateur final. On le qualifie de principal car c'est lui qui importe tous les autres modules nécessités par le programme. On l'appelle parfois le point d'entrée de l'application.

L'environnement principal peut prendre diverses formes :

— l'environnement d'une invite de commande interactive :

```
>>> __name__
'__main__'
```

— le module passé directement en tant que fichier à la commande de l'interpréteur Python :

```
$ python helloworld.py
Hello, world!
```

— le module ou paquet passé à l'interpréteur avec l'option `-m` :

```
$ python -m tarfile
usage: tarfile.py [-h] [-v] (...)
```

— le code lu par l'interpréteur depuis l'entrée standard :

```
$ echo "import this" | python
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
...
```

— le code passé à l'interpréteur avec l'option `-c` :

```
$ python -c "import this"
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
...
```

Dans chacun de ces cas, l'attribut `__name__` du module principal est mis à `'__main__'`.

Un module peut donc savoir s'il est exécuté dans l'environnement principal en vérifiant son `__name__`, ce qui permet typiquement d'exécuter du code lorsque le module est initialisé d'une manière autre que l'importation :

```
if __name__ == '__main__':
    # Execute when the module is not initialized from an import statement.
    ...
```

Voir aussi

Pour plus de détails sur la manière dont `__name__` est défini dans les divers cas, voir la section tut-modules dans le tutoriel.

Utilisation idiomatique

Il arrive qu'un module contienne du code qui ne doit s'exécuter que lorsque le module est utilisé comme script. On peut penser à l'analyse des arguments passés en ligne de commande, ou bien à la lecture de l'entrée standard. Il ne faudrait pas que ces opérations soient effectuées lorsque le module est importé depuis un autre module, comme pour les tests.

C'est dans ces situations que sert la construction `if __name__ == '__main__'`. Le code mis à l'intérieur du bloc n'est exécuté que si le module est dans l'environnement principal.

Putting as few statements as possible in the block below `if __name__ == '__main__'` can improve code clarity and correctness. Most often, a function named `main` encapsulates the program's primary behavior :

```
# echo.py

import shlex
import sys

def echo(phrase: str) -> None:
   """A dummy wrapper around print."""
   # for demonstration purposes, you can imagine that there is some
   # valuable and reusable logic inside this function
   print(phrase)

def main() -> int:
    """Echo the input arguments to standard output"""
    phrase = shlex.join(sys.argv)
    echo(phrase)
    return 0
```

(suite sur la page suivante)

(suite de la page précédente)

```
if __name__ == '__main__':
    sys.exit(main())  # next section explains the use of sys.exit
```

Si, dans le module ci-dessus, le code de `main` était placé directement dans le `if __name__ == '__main__'`, la variable `phrase` serait globale, et d'autres fonctions pourraient s'en servir par erreur à la place d'une variable locale. Encapsuler le code dans la fonction `main` évite cet ennui.

De plus, la fonction `echo` est elle-même séparée du reste, et on peut l'importer dans un autre module. Le fait d'importer `echo.py` définit les fonctions `main` et `echo`, mais n'en appelle aucune, puisque `__name__ != '__main__'`.

Considérations liées à l'empaquetage

Les fonctions `main` servent souvent à créer des outils qui s'exécutent en ligne de commande. Les scripts sont ajoutés au système à l'aide de points d'entrée, qui demandent à pip de créer un exécutable. Il le fait en insérant un appel à la fonction à l'intérieur d'un modèle prédéfini où la valeur qu'elle renvoie est passée directement à *sys.exit()* :

```
sys.exit(main())
```

Puisque l'appel de `main` est encapsulé dans *sys.exit()*, `main` doit renvoyer une valeur qui convienne comme argument à *sys.exit()*, par exemple un code de retour sous forme d'entier. La valeur `None` est également acceptée, et c'est d'ailleurs celle que renvoie la fonction si elle se termine sans rencontrer d'instruction `return`.

By proactively following this convention ourselves, our module will have the same behavior when run directly (i.e. `python echo.py`) as it will have if we later package it as a console script entry-point in a pip-installable package.

En particulier, mieux vaut éviter de renvoyer une chaîne de caractères depuis la fonction `main`. En effet, si *sys.exit()* reçoit une chaîne, elle l'interprète comme un message d'erreur, qu'elle affiche sur *sys.stderr* avant de terminer le programme avec le code de retour 1 (erreur). L'exemple de `echo.py` ci-dessus montre la pratique recommandée.

Voir aussi

Le guide de l'empaquetage en Python (en anglais) contient plusieurs tutoriels et documents de référence autour de la distribution et de l'installation de paquets Python avec des outils modernes.

30.5.2 Le fichier `__main__.py` dans les paquets Python

Si vous n'êtes pas familier des paquets Python, lisez tut-packages dans le tutoriel. Un fichier `__main__.py` permet à un paquet de définir une interface en ligne de commande. Prenons pour exemple un paquet nommé *bandclass* :

```
bandclass
  ├── __init__.py
  ├── __main__.py
  └── student.py
```

Le fichier `__main__.py` qu'il contient s'exécute lorsque le paquet est appelé depuis la ligne de commande avec l'option `-m` de l'interpréteur, comme ceci :

```
$ python -m bandclass
```

Cette commande lance l'exécution de `__main__.py`. Il vous appartient, en tant que concepteur du paquet, de déterminer ce qu'elle doit faire. Dans notre exemple, elle pourrait rechercher un étudiant dans une base de données :

```
# bandclass/__main__.py

import sys
from .student import search_students
```

(suite sur la page suivante)

(suite de la page précédente)

```
student_name = sys.argv[1] if len(sys.argv) >= 2 else ''
print(f'Found student: {search_students(student_name)}')
```

Remarquez l'importation `from .student import search_students`. Le point avant `student` sert à rendre le chemin `student` relatif à la position du module qui l'importe. Pour plus d'informations, voir intra-package-references dans la section tut-modules du tutoriel.

Utilisation idiomatique

The content of `__main__.py` typically isn't fenced with an `if __name__ == '__main__'` block. Instead, those files are kept short and import functions to execute from other modules. Those other modules can then be easily unit-tested and are properly reusable.

Cependant, un `if __name__ == '__main__'`, s'il est présent dans le `__main__.py`, fonctionne correctement. En effet, si `__main__.py` est importé depuis autre module, son attribut `__name__` contient, avant `__main__`, le nom du paquet dont il fait partie :

```
>>> import asyncio.__main__
>>> asyncio.__main__.__name__
'asyncio.__main__'
```

This won't work for `__main__.py` files in the root directory of a `.zip` file though. Hence, for consistency, a minimal `__main__.py` without a `__name__` check is preferred.

Voir aussi

See *venv* for an example of a package with a minimal `__main__.py` in the standard library. It doesn't contain a `if __name__ == '__main__'` block. You can invoke it with `python -m venv [directory]`.

La documentation du module *runpy* fournit une description complète de l'option `-m` de l'interpréteur.

Le module *zipapp* exécute des applications emballées dans une archive ZIP. Dans ce cas, l'interpréteur recherche un fichier `__main__.py` à la racine de l'archive.

30.5.3 `import __main__`

Quel que soit le module principal d'un programme, les autres modules peuvent accéder à l'*espace de nommage* dans lequel il s'exécute en important le module spécial `__main__`. Celui-ci ne correspond pas forcément à un fichier `__main__.py`. Il s'agit simplement du module qui a reçu le nom `'__main__'`.

Voici un exemple d'utilisation du module `__main__` :

```
# namely.py

import __main__

def did_user_define_their_name():
    return 'my_name' in dir(__main__)

def print_user_name():
    if not did_user_define_their_name():
        raise ValueError('Define the variable `my_name`!')

    print(__main__.my_name)
```

Ce code s'utilise comme ceci :

```
# start.py

import sys

from namely import print_user_name

# my_name = "Dinsdale"

def main():
    try:
        print_user_name()
    except ValueError as ve:
        return str(ve)

if __name__ == "__main__":
    sys.exit(main())
```

Le programme ci-dessus donne la sortie :

```
$ python start.py
Define the variable `my_name`!
```

Son code de retour est 1, ce qui signifie une erreur. En supprimant la marque de commentaire en début de ligne `my_name = "Dinsdale"`, le programme est corrigé, et renvoie au système le code 0 car il n'y a plus d'erreur :

```
$ python start.py
Dinsdale
```

On pourrait s'attendre à un problème au moment de l'importation de `__main__` : cela ne provoque-t-il pas l'exécution anticipée du code de script sous `if __name__ == '__main__'` dans le module principal `start` ?

Python inserts an empty `__main__` module in *sys.modules* at interpreter startup, and populates it by running top-level code. In our example this is the `start` module which runs line by line and imports `namely`. In turn, `namely` imports `__main__` (which is really `start`). That's an import cycle ! Fortunately, since the partially populated `__main__` module is present in *sys.modules*, Python passes that to `namely`. See Special considerations for __main__ in the import system's reference for details on how this works.

L'interpréteur interactif est un autre environnement d'exécution principal possible. Toute variable qui y est définie appartient à l'espace de nommage `__main__` :

```
>>> import namely
>>> namely.did_user_define_their_name()
False
>>> namely.print_user_name()
Traceback (most recent call last):
...
ValueError: Define the variable `my_name`!
>>> my_name = 'Jabberwocky'
>>> namely.did_user_define_their_name()
True
>>> namely.print_user_name()
Jabberwocky
```

Le module `__main__` est notamment employé dans les implémentations de *pdb* et *rlcompleter*.

30.6 warnings --- Warning control

Code source : Lib/warnings.py

Les messages d'avertissement sont généralement émis dans les situations où il est utile d'alerter l'utilisateur d'un problème dans un programme, mais qu'il n'est pas justifié de lever une exception et de le terminer. Par exemple, on peut vouloir émettre un avertissement lorsqu'un programme utilise un module obsolète.

Les développeurs Python émettent des avertissements en appelant la fonction `warn()` définie dans ce module. (Les développeurs C utilisent `PyErr_WarnEx()` ; voir exceptionhandling pour plus d'informations).

Les messages d'avertissement sont normalement écrits sur `sys.stderr`, mais leurs effets peuvent être modifiés, il est possible d'ignorer tous les avertissements ou au contraire les transformer en exceptions. L'effet des avertissements peut varier selon la *catégorie d'avertissement*, de son texte et d'où il est émis. Les répétitions du même avertissement depuis une même source sont généralement ignorées.

La gestion des avertissements se fait en deux étapes : premièrement, chaque fois qu'un avertissement est émis, le module détermine si un message doit être émis ou non ; ensuite, si un message doit être émis, il est formaté et affiché en utilisant une fonction qui peut être définie par l'utilisateur.

Un *filtre* (une séquence de règles) est utilisé pour décider si un message d'avertissement doit être émis ou non. Des règles peuvent être ajoutées au filtre en appelant `filterwarnings()` et remises à leur état par défaut en appelant `resetwarnings()`.

L'affichage des messages d'avertissement se fait en appelant la fonction `showwarning()`, qui peut être redéfinie ; l'implémentation par défaut formate le message en appelant `formatwarning()`, qui peut également être réutilisée par une implémentation personnalisée.

Voir aussi

`logging.captureWarnings()` vous permet de gérer tous les avertissements avec l'infrastructure de journalisation standard.

30.6.1 Catégories d'avertissement

Il existe un certain nombre d'exceptions natives qui représentent des catégories d'avertissement. Cette catégorisation est utile pour filtrer les groupes d'avertissements.

Bien qu'il s'agisse techniquement d'exceptions, les *exceptions natives* sont documentées ici, parce qu'elles appartiennent conceptuellement au mécanisme des avertissements.

Le code utilisateur peut définir des catégories d'avertissement supplémentaires en héritant l'une des catégories d'avertissement standard. Une catégorie d'avertissement doit toujours hériter de la classe `Warning`.

Les classes de catégories d'avertissement suivantes sont actuellement définies :

Classe	Description
`Warning`	Il s'agit de la classe de base de toutes les classes de catégories d'avertissement. C'est une sous-classe de `Exception`.
`UserWarning`	Catégorie par défaut pour `warn()`.
`DeprecationWarning`	Catégorie de base pour les avertissements sur les fonctionnalités obsolètes lorsque ces avertissements sont destinés à d'autres développeurs Python (ignorées par défaut, sauf si elles proviennent de `__main__`).
`SyntaxWarning`	Catégorie de base pour les avertissements concernant les syntaxes douteuses.
`RuntimeWarning`	Catégorie de base pour les avertissements concernant les fonctionnalités douteuses à l'exécution.
`FutureWarning`	Catégorie de base pour les avertissements concernant les fonctionnalités obsolètes lorsque ces avertissements sont destinés aux utilisateurs finaux des programmes écrits en Python.
`PendingDeprecationWarning`	Catégorie de base pour les avertissements concernant les fonctionnalités qui seront obsolètes dans le futur (ignorée par défaut).
`ImportWarning`	Catégorie de base pour les avertissements déclenchés lors de l'importation d'un module (ignoré par défaut).
`UnicodeWarning`	Catégorie de base pour les avertissements relatifs à Unicode.
`BytesWarning`	Catégorie de base pour les avertissements relatifs à `bytes` et `bytearray`.
`ResourceWarning`	Catégorie de base pour les avertissements relatifs à l'utilisation des ressources (ignorés par défaut).

Modifié dans la version 3.7 : Avant, la différence entre `DeprecationWarning` et `FutureWarning` était que l'un était dédié aux fonctionnalités retirées, et l'autre aux fonctionnalités modifiées. La différence aujourd'hui est plutôt leur audience et la façon dont ils sont traités par les filtres d'avertissement par défaut.

30.6.2 Le filtre des avertissements

Le filtre des avertissements contrôle si les avertissements sont ignorés, affichés ou transformés en erreurs (ce qui lève une exception).

Conceptuellement, le filtre d'avertissements maintient une liste ordonnée d'entrées ; chaque avertissement est comparé à chaque entrée de la liste jusqu'à ce qu'une correspondance soit trouvée ; l'entrée détermine l'action à effectuer. Chaque entrée est un quintuplet de la forme (*action*, *message*, *catégorie*, *module*, *lineno*), où :

— *action* est l'une des chaînes de caractères suivantes :

Valeur	Action
`"defaul`	affiche la première occurrence des avertissements correspondants pour chaque emplacement (module + numéro de ligne) où l'avertissement est émis
`"error"`	transforme les avertissements correspondants en exceptions
`"ignore`	ignore les avertissements correspondants
`"always`	affiche toujours les avertissements correspondants
`"all"`	alias to "always"
`"module`	affiche la première occurrence des avertissements correspondants pour chaque module où l'avertissement est émis (quel que soit le numéro de ligne)
`"once"`	n'affiche que la première occurrence des avertissements correspondants, quel que soit l'endroit où ils se trouvent

— *message* is a string containing a regular expression that the start of the warning message must match, case-insensitively. In `-W` and `PYTHONWARNINGS`, *message* is a literal string that the start of the warning message must contain (case-insensitively), ignoring any whitespace at the start or end of *message*.

— *category* est une classe (une sous-classe de `Warning`) dont la catégorie d'avertissement doit être une sous-classe afin de correspondre.

— *module* is a string containing a regular expression that the start of the fully qualified module name must match, case-sensitively. In `-W` and `PYTHONWARNINGS`, *module* is a literal string that the fully qualified module name must be equal to (case-sensitively), ignoring any whitespace at the start or end of *module*.
— *lineno* est le numéro de ligne d'où l'avertissement doit provenir, ou `0` pour correspondre à tous les numéros de ligne.

Puisque que la classe `Warning` hérite de la classe `Exception`, pour transformer un avertissement en erreur, il suffit de lever `category(message)`.

Si un avertissement est signalé et ne correspond à aucun filtre enregistré, l'action `default` est appliquée (d'où son nom).

Repeated Warning Suppression Criteria

The filters that suppress repeated warnings apply the following criteria to determine if a warning is considered a repeat :

— `"default"` : A warning is considered a repeat only if the (*message*, *category*, *module*, *lineno*) are all the same.
— `"module"` : A warning is considered a repeat if the (*message*, *category*, *module*) are the same, ignoring the line number.
— `"once"` : A warning is considered a repeat if the (*message*, *category*) are the same, ignoring the module and line number.

Rédaction de filtres d'avertissement

Le filtre des avertissements est initialisé par les options `-W` passées à la ligne de commande de l'interpréteur Python et la variable d'environnement `PYTHONWARNINGS`. L'interpréteur enregistre les arguments de toutes les entrées fournies sans interprétation dans `sys.warnoptions`; le module `warnings` les analyse lors de la première importation (les options invalides sont ignorées, et un message d'erreur est envoyé à `sys.stderr`).

Les filtres d'avertissement individuels sont décrits sous la forme d'une séquence de champs séparés par des deux-points :

```
action:message:category:module:line
```

La signification de chacun de ces champs est décrite dans *Le filtre des avertissements*. Plusieurs filtres peuvent être écrits en une seule ligne (comme pour `PYTHONWARNINGS`), ils sont dans ce cas séparés par des virgules, et les filtres listés plus en dernier ont priorité sur ceux qui les précèdent (car ils sont appliqués de gauche à droite, et les filtres les plus récemment appliqués ont priorité sur les précédents).

Les filtres d'avertissement couramment utilisés s'appliquent à tous les avertissements, aux avertissements d'une catégorie particulière ou aux avertissements émis par certains modules ou paquets. Quelques exemples :

```
default                      # Show all warnings (even those ignored by default)
ignore                       # Ignore all warnings
error                        # Convert all warnings to errors
error::ResourceWarning       # Treat ResourceWarning messages as errors
default::DeprecationWarning  # Show DeprecationWarning messages
ignore,default:::mymodule    # Only report warnings triggered by "mymodule"
error:::mymodule             # Convert warnings to errors in "mymodule"
```

Filtre d'avertissement par défaut

Par défaut, Python installe plusieurs filtres d'avertissement, qui peuvent être outrepassés par l'option `-W` en ligne de commande, la variable d'environnement `PYTHONWARNINGS` et les appels à `filterwarnings()`.

Dans les versions standard publiées de Python, le filtre d'avertissement par défaut a les entrées suivantes (par ordre de priorité) :

```
default::DeprecationWarning:__main__
ignore::DeprecationWarning
```

(suite sur la page suivante)

(suite de la page précédente)

```
ignore::PendingDeprecationWarning
ignore::ImportWarning
ignore::ResourceWarning
```

Dans les versions de débogage, la liste des filtres d'avertissement par défaut est vide.

Modifié dans la version 3.2 : `DeprecationWarning` est maintenant ignoré par défaut en plus de `PendingDeprecationWarning`.

Modifié dans la version 3.7 : `DeprecationWarning` est à nouveau affiché par défaut lorsqu'il provient directement de `__main__`.

Modifié dans la version 3.7 : `BytesWarning` n'apparaît plus dans la liste de filtres par défaut et est configuré via `sys.warnoptions` lorsque l'option `-b` est donnée deux fois.

Outrepasser le filtre par défaut

Les développeurs d'applications écrites en Python peuvent souhaiter cacher *tous* les avertissements Python à leurs utilisateurs, et ne les afficher que lorsqu'ils exécutent des tests ou travaillent sur l'application. L'attribut `sys.warnoptions` utilisé pour passer les configurations de filtre à l'interpréteur peut être utilisé comme marqueur pour indiquer si les avertissements doivent être ou non désactivés :

```
import sys

if not sys.warnoptions:
    import warnings
    warnings.simplefilter("ignore")
```

Les développeurs d'exécuteurs de test pour le code Python sont invités à s'assurer que *tous* les avertissements sont affichés par défaut pour le code en cours de test, en utilisant par exemple :

```
import sys

if not sys.warnoptions:
    import os, warnings
    warnings.simplefilter("default") # Change the filter in this process
    os.environ["PYTHONWARNINGS"] = "default" # Also affect subprocesses
```

Enfin, les développeurs d'interpréteurs de commandes interactifs qui exécutent du code utilisateur dans un espace de nommage autre que `__main__` sont invités à s'assurer que les messages `DeprecationWarning` sont rendus visibles par défaut, en utilisant le code suivant (où `user_ns` est le module utilisé pour exécuter le code entré interactivement) :

```
import warnings
warnings.filterwarnings("default", category=DeprecationWarning,
                                   module=user_ns.get("__name__"))
```

30.6.3 Suppression temporaire des avertissements

Si vous utilisez un code dont vous savez qu'il va déclencher un avertissement, comme une fonction obsolète, mais que vous ne voulez pas voir l'avertissement (même si les avertissements ont été explicitement configurés via la ligne de commande), alors il est possible de supprimer l'avertissement en utilisant le gestionnaire de contexte `catch_warnings` :

```
import warnings

def fxn():
    warnings.warn("deprecated", DeprecationWarning)
```

(suite sur la page suivante)

(suite de la page précédente)

```
with warnings.catch_warnings():
    warnings.simplefilter("ignore")
    fxn()
```

Dans le gestionnaire de contexte, tous les avertissements sont simplement ignorés. Ceci vous permet d'utiliser du code déclaré obsolète sans voir l'avertissement tout en ne supprimant pas l'avertissement pour un autre code qui pourrait ne pas être conscient de son utilisation de code déprécié. Remarque : ceci ne peut être garanti que dans une application utilisant un seul fil d'exécution. Si deux ou plusieurs *threads* utilisent le gestionnaire de contexte `catch_warnings` en même temps, le comportement est indéfini.

30.6.4 Tester les avertissements

Pour tester les avertissements générés par le code, utilisez le gestionnaire de contexte `catch_warnings`. Avec lui, vous pouvez temporairement modifier le filtre d'avertissements pour faciliter votre test. Par exemple, procédez comme suit pour capturer tous les avertissements levés à vérifier :

```
import warnings

def fxn():
    warnings.warn("deprecated", DeprecationWarning)

with warnings.catch_warnings(record=True) as w:
    # Cause all warnings to always be triggered.
    warnings.simplefilter("always")
    # Trigger a warning.
    fxn()
    # Verify some things
    assert len(w) == 1
    assert issubclass(w[-1].category, DeprecationWarning)
    assert "deprecated" in str(w[-1].message)
```

Vous pouvez aussi faire en sorte que tous les avertissements soient des exceptions en utilisant `error` au lieu de `always`. Il faut savoir que si un avertissement a déjà été émis à cause d'une règle `once` ou `default`, quel que soit le filtre activé, l'avertissement ne sera pas revu à moins que le registre des avertissements lié à l'avertissement ait été vidé.

A sa sortie, le gestionnaire de contexte restaure le filtre des avertissements dans l'état où il était au démarrage du contexte. Cela empêche les tests de changer le filtre d'avertissements de manière inattendue entre les tests et d'aboutir à des résultats de test indéterminés. La fonction `showwarning()` du module est également restaurée à sa valeur originale. Remarque : ceci ne peut être garanti que dans une application *mono-threadées*. Si deux ou plusieurs fils d'exécution utilisent le gestionnaire de contexte `catch_warnings` en même temps, le comportement est indéfini.

Lorsque vous testez plusieurs opérations qui provoquent le même type d'avertissement, il est important de les tester d'une manière qui confirme que chaque opération provoque un nouvel avertissement (par exemple, définissez les avertissements comme exceptions et vérifiez que les opérations provoquent des exceptions, vérifiez que la longueur de la liste des avertissements continue à augmenter après chaque opération, ou bien supprimez les entrées précédentes de la liste des avertissements avant chaque nouvelle opération).

30.6.5 Mise à jour du code pour les nouvelles versions des dépendances

Les catégories d'avertissement qui intéressent principalement les développeurs Python (plutôt que les utilisateurs finaux d'applications écrites en Python) sont ignorées par défaut.

Notamment, cette liste "ignorés par défaut" inclut `DeprecationWarning` (pour chaque module sauf `__main__`), ce qui signifie que les développeurs doivent s'assurer de tester leur code avec des avertissements généralement ignorés rendus visibles afin de recevoir des notifications rapides des changements d'API (que ce soit dans la bibliothèque standard ou les paquets tiers).

Dans le cas idéal, le code dispose d'une suite de tests appropriée, et le testeur se charge d'activer implicitement tous les avertissements lors de l'exécution des tests (le testeur fourni par le module `unittest` le fait).

Dans des cas moins idéaux, l'utilisation de d'interfaces obsolète peut être testé en passant `-Wd` à l'interpréteur Python (c'est une abréviation pour `-W default`) ou en définissant `PYTHONWARNINGS=default` dans l'environnement. Ceci permet la gestion par défaut de tous les avertissements, y compris ceux qui sont ignorés par défaut. Pour changer l'action prise pour les avertissements rencontrés, vous pouvez changer quel argument est passé à `-W` (par exemple `-W error`). Voir l'option `-W` pour plus de détails sur ce qui est possible.

30.6.6 Fonctions disponibles

`warnings.`**`warn`**(*message*, *category=None*, *stacklevel=1*, *source=None*, **, skip_file_prefixes=()*)

Émet, ignore, ou transforme en exception un avertissement. L'argument *category*, s'il est donné, doit être une classe de *warning category class*; et vaut par défaut `UserWarning`. Aussi *message* peut être une instance de `Warning`, auquel cas *category* sera ignoré et `message.__class__` sera utilisé. Dans ce cas, le texte du message sera `str(message)`. Cette fonction lève une exception si cet avertissement particulier émis est transformé en erreur par le *filtre des avertissements*. L'argument *stacklevel* peut être utilisé par les fonctions *wrapper* écrites en Python, comme ceci :

```
def deprecated_api(message):
    warnings.warn(message, DeprecationWarning, stacklevel=2)
```

This makes the warning refer to `deprecated_api`'s caller, rather than to the source of `deprecated_api` itself (since the latter would defeat the purpose of the warning message).

The *skip_file_prefixes* keyword argument can be used to indicate which stack frames are ignored when counting stack levels. This can be useful when you want the warning to always appear at call sites outside of a package when a constant *stacklevel* does not fit all call paths or is otherwise challenging to maintain. If supplied, it must be a tuple of strings. When prefixes are supplied, stacklevel is implicitly overridden to be `max(2, stacklevel)`. To cause a warning to be attributed to the caller from outside of the current package you might write :

```
# example/lower.py
_warn_skips = (os.path.dirname(__file__),)

def one_way(r_luxury_yacht=None, t_wobbler_mangrove=None):
    if r_luxury_yacht:
        warnings.warn("Please migrate to t_wobbler_mangrove=.",
                      skip_file_prefixes=_warn_skips)

# example/higher.py
from . import lower

def another_way(**kw):
    lower.one_way(**kw)
```

This makes the warning refer to both the `example.lower.one_way()` and `package.higher.another_way()` call sites only from calling code living outside of `example` package.

source, s'il est fourni, est l'objet détruit qui a émis un `ResourceWarning`.

Modifié dans la version 3.6 : Ajout du paramètre *source*.

Modifié dans la version 3.12 : Added *skip_file_prefixes*.

`warnings.`**`warn_explicit`**(*message*, *category*, *filename*, *lineno*, *module=None*, *registry=None*, *module_globals=None*, *source=None*)

Il s'agit d'une interface de bas niveau pour la fonctionnalité de `warn()`, en passant explicitement le message, la catégorie, le nom de fichier et le numéro de ligne, et éventuellement le nom du module et le registre (qui devrait être le dictionnaire `__warningregistry__` du module). Le nom de module par défaut est le nom de fichier sans `.py`; si aucun registre n'est passé, l'avertissement n'est jamais supprimé. *message* doit être une

chaîne de caractères et *category* une sous-classe de `Warning` ou *message* peut être une instance de `Warning`, auquel cas *category* sera ignoré.

module_globals, s'il est fourni, doit être l'espace de nommage global utilisé par le code pour lequel l'avertissement est émis. (Cet argument est utilisé pour afficher les sources des modules trouvés dans les fichiers zip ou d'autres sources d'importation hors du système de fichiers).

source, s'il est fourni, est l'objet détruit qui a émis un `ResourceWarning`.

Modifié dans la version 3.6 : Ajout du paramètre *source*.

`warnings.`**`showwarning`**(*message*, *category*, *filename*, *lineno*, *file=None*, *line=None*)

Écrit un avertissement dans un fichier. L'implémentation par défaut appelle `formatwarning(message, category, filename, lineno, line)` et écrit la chaîne résultante dans *file*, qui par défaut est `sys.stderr`. Vous pouvez remplacer cette fonction par n'importe quel appelable en l'affectant à `warnings.showwarning`. *line* est une ligne de code source à inclure dans le message d'avertissement ; si *line* n'est pas fourni, `showwarning()` essaiera de lire la ligne spécifiée par *filename* et *lineno*.

`warnings.`**`formatwarning`**(*message*, *category*, *filename*, *lineno*, *line=None*)

Formate un avertissement de la manière standard. Ceci renvoie une chaîne pouvant contenir des retours à la ligne se termine par un retour à la ligne. *line* est une ligne de code source à inclure dans le message d'avertissement ; si *line* n'est pas fourni, `formatwarning()` essaiera de lire la ligne spécifiée par *filename* et *lineno*.

`warnings.`**`filterwarnings`**(*action*, *message=''*, *category=Warning*, *module=''*, *lineno=0*, *append=False*)

Insère une entrée dans la liste de *warning filter specifications*. L'entrée est insérée à l'avant par défaut ; si *append* est vrai, elle est insérée à la fin. Il vérifie le type des arguments, compile les expressions régulières *message* et *module*, et les insère sous forme de *n*-uplet dans la liste des filtres d'avertissements. Les entrées plus proches du début de la liste ont priorité sur les entrées plus loin dans la liste. Les arguments omis ont par défaut une valeur qui correspond à tout.

`warnings.`**`simplefilter`**(*action*, *category=Warning*, *lineno=0*, *append=False*)

Insère une entrée simple dans la liste de *spécifications du filtre d'avertissements*. La signification des paramètres de fonction est la même que pour `filterwarnings()`, mais les expressions régulières ne sont pas nécessaires car le filtre inséré correspond toujours à n'importe quel message dans n'importe quel module tant que la catégorie et le numéro de ligne correspondent.

`warnings.`**`resetwarnings`**()

Réinitialise le filtre des avertissements. Ceci supprime l'effet de tous les appels précédents à `filterwarnings()`, y compris celui de l'option `-W` des options de ligne de commande et des appels à `simplefilter()`.

`@warnings.`**`deprecated`**(*msg*, ***, *category=DeprecationWarning*, *stacklevel=1*)

Decorator to indicate that a class, function or overload is deprecated.

When this decorator is applied to an object, deprecation warnings may be emitted at runtime when the object is used. *static type checkers* will also generate a diagnostic on usage of the deprecated object.

Utilisation :

```
from warnings import deprecated
from typing import overload

@deprecated("Use B instead")
class A:
    pass

@deprecated("Use g instead")
def f():
    pass

@overload
@deprecated("int support is deprecated")
```

(suite sur la page suivante)

(suite de la page précédente)

```
def g(x: int) -> int: ...
@overload
def g(x: str) -> int: ...
```

The warning specified by *category* will be emitted at runtime on use of deprecated objects. For functions, that happens on calls; for classes, on instantiation and on creation of subclasses. If the *category* is `None`, no warning is emitted at runtime. The *stacklevel* determines where the warning is emitted. If it is `1` (the default), the warning is emitted at the direct caller of the deprecated object; if it is higher, it is emitted further up the stack. Static type checker behavior is not affected by the *category* and *stacklevel* arguments.

The deprecation message passed to the decorator is saved in the `__deprecated__` attribute on the decorated object. If applied to an overload, the decorator must be after the `@overload` decorator for the attribute to exist on the overload as returned by `typing.get_overloads()`.

Ajouté dans la version 3.13 : See **PEP 702**.

30.6.7 Gestionnaires de contexte disponibles

class warnings.**catch_warnings**(*, *record=False*, *module=None*, *action=None*, *category=Warning*, *lineno=0*, *append=False*)

Un gestionnaire de contexte qui copie et, à la sortie, restaure le filtre des avertissements et la fonction `showwarning()`. Si l'argument *record* est `False` (par défaut), le gestionnaire de contexte retourne `None` en entrant. Si *record* est `True`, une liste est renvoyée qui est progressivement remplie d'objets comme vus par une fonction custom `showwarning' (qui supprime également la sortie vers ``sys.stdout`()`). Chaque objet de la liste a des attributs avec les mêmes noms que les arguments de `showwarning()`.

L'argument *module* prend un module qui sera utilisé à la place du module renvoyé lors de l'importation `warnings` dont le filtre sera protégé. Cet argument existe principalement pour tester le module `warnings` lui-même.

If the *action* argument is not `None`, the remaining arguments are passed to `simplefilter()` as if it were called immediately on entering the context.

See *Le filtre des avertissements* for the meaning of the *category* and *lineno* parameters.

> **Note**
>
> Le gestionnaire `catch_warnings` fonctionne en remplaçant puis en restaurant plus tard la fonction `showwarning()` du module et la liste interne des spécifications du filtre. Cela signifie que le gestionnaire de contexte modifie l'état global et n'est donc pas prévisible avec plusieurs fils d'exécution.

Modifié dans la version 3.11 : Added the *action*, *category*, *lineno*, and *append* parameters.

30.7 `dataclasses` --- Data Classes

Code source : Lib/dataclasses.py

This module provides a decorator and functions for automatically adding generated *special methods* such as `__init__()` and `__repr__()` to user-defined classes. It was originally described in **PEP 557**.

Les variables membres à utiliser dans ces méthodes générées sont définies en utilisant les annotations de type **PEP 526**. Par exemple :

```
from dataclasses import dataclass
```

(suite sur la page suivante)

(suite de la page précédente)

```
@dataclass
class InventoryItem:
    """Class for keeping track of an item in inventory."""
    name: str
    unit_price: float
    quantity_on_hand: int = 0

    def total_cost(self) -> float:
        return self.unit_price * self.quantity_on_hand
```

will add, among other things, a `__init__()` that looks like :

```
def __init__(self, name: str, unit_price: float, quantity_on_hand: int = 0):
    self.name = name
    self.unit_price = unit_price
    self.quantity_on_hand = quantity_on_hand
```

Note that this method is automatically added to the class : it is not directly specified in the `InventoryItem` definition shown above.

Ajouté dans la version 3.7.

30.7.1 Classe de données

`@dataclasses.`**`dataclass`**(*, *init=True*, *repr=True*, *eq=True*, *order=False*, *unsafe_hash=False*, *frozen=False*, *match_args=True*, *kw_only=False*, *slots=False*, *weakref_slot=False*)

This function is a *decorator* that is used to add generated *special methods* to classes, as described below.

The `@dataclass` decorator examines the class to find `field`s. A `field` is defined as a class variable that has a *type annotation*. With two exceptions described below, nothing in `@dataclass` examines the type specified in the variable annotation.

L'ordre des paramètres des méthodes générées est celui d'apparition des champs dans la définition de la classe.

The `@dataclass` decorator will add various "dunder" methods to the class, described below. If any of the added methods already exist in the class, the behavior depends on the parameter, as documented below. The decorator returns the same class that it is called on ; no new class is created.

If `@dataclass` is used just as a simple decorator with no parameters, it acts as if it has the default values documented in this signature. That is, these three uses of `@dataclass` are equivalent :

```
@dataclass
class C:
    ...

@dataclass()
class C:
    ...

@dataclass(init=True, repr=True, eq=True, order=False, unsafe_hash=False,␣
↪frozen=False,
           match_args=True, kw_only=False, slots=False, weakref_slot=False)
class C:
    ...
```

The parameters to `@dataclass` are :

— *init* : If true (the default), a `__init__()` method will be generated.

If the class already defines `__init__()`, this parameter is ignored.

— *repr* : If true (the default), a `__repr__()` method will be generated. The generated repr string will have the class name and the name and repr of each field, in the order they are defined in the class. Fields that are marked as being excluded from the repr are not included. For example : `InventoryItem(name='widget', unit_price=3.0, quantity_on_hand=10)`.

If the class already defines `__repr__()`, this parameter is ignored.

— *eq* : If true (the default), an `__eq__()` method will be generated. This method compares the class as if it were a tuple of its fields, in order. Both instances in the comparison must be of the identical type.

If the class already defines `__eq__()`, this parameter is ignored.

— *order* : If true (the default is `False`), `__lt__()`, `__le__()`, `__gt__()`, and `__ge__()` methods will be generated. These compare the class as if it were a tuple of its fields, in order. Both instances in the comparison must be of the identical type. If *order* is true and *eq* is false, a `ValueError` is raised.

If the class already defines any of `__lt__()`, `__le__()`, `__gt__()`, or `__ge__()`, then `TypeError` is raised.

— *unsafe_hash* : If `False` (the default), a `__hash__()` method is generated according to how *eq* and *frozen* are set.

`__hash__()` is used by built-in `hash()`, and when objects are added to hashed collections such as dictionaries and sets. Having a `__hash__()` implies that instances of the class are immutable. Mutability is a complicated property that depends on the programmer's intent, the existence and behavior of `__eq__()`, and the values of the *eq* and *frozen* flags in the `@dataclass` decorator.

By default, `@dataclass` will not implicitly add a `__hash__()` method unless it is safe to do so. Neither will it add or change an existing explicitly defined `__hash__()` method. Setting the class attribute `__hash__ = None` has a specific meaning to Python, as described in the `__hash__()` documentation.

If `__hash__()` is not explicitly defined, or if it is set to `None`, then `@dataclass` *may* add an implicit `__hash__()` method. Although not recommended, you can force `@dataclass` to create a `__hash__()` method with `unsafe_hash=True`. This might be the case if your class is logically immutable but can still be mutated. This is a specialized use case and should be considered carefully.

Here are the rules governing implicit creation of a `__hash__()` method. Note that you cannot both have an explicit `__hash__()` method in your dataclass and set `unsafe_hash=True`; this will result in a `TypeError`.

If *eq* and *frozen* are both true, by default `@dataclass` will generate a `__hash__()` method for you. If *eq* is true and *frozen* is false, `__hash__()` will be set to `None`, marking it unhashable (which it is, since it is mutable). If *eq* is false, `__hash__()` will be left untouched meaning the `__hash__()` method of the superclass will be used (if the superclass is `object`, this means it will fall back to id-based hashing).

— *frozen* : If true (the default is `False`), assigning to fields will generate an exception. This emulates read-only frozen instances. If `__setattr__()` or `__delattr__()` is defined in the class, then `TypeError` is raised. See the discussion below.

— *match_args* : If true (the default is `True`), the `__match_args__` tuple will be created from the list of parameters to the generated `__init__()` method (even if `__init__()` is not generated, see above). If false, or if `__match_args__` is already defined in the class, then `__match_args__` will not be generated.

Ajouté dans la version 3.10.

— *kw_only* : If true (the default value is `False`), then all fields will be marked as keyword-only. If a field is marked as keyword-only, then the only effect is that the `__init__()` parameter generated from a keyword-only field must be specified with a keyword when `__init__()` is called. There is no effect on any other aspect of dataclasses. See the *parameter* glossary entry for details. Also see the `KW_ONLY` section.

Ajouté dans la version 3.10.

— *slots* : If true (the default is `False`), `__slots__` attribute will be generated and new class will be returned instead of the original one. If `__slots__` is already defined in the class, then `TypeError` is raised.

Avertissement

> Passing parameters to a base class `__init_subclass__()` when using `slots=True` will result in a `TypeError`. Either use `__init_subclass__` with no parameters or use default values as a workaround. See gh-91126 for full details.

Ajouté dans la version 3.10.

Modifié dans la version 3.11 : If a field name is already included in the `__slots__` of a base class, it will not be included in the generated `__slots__` to prevent overriding them. Therefore, do not use `__slots__` to retrieve the field names of a dataclass. Use `fields()` instead. To be able to determine inherited slots, base class `__slots__` may be any iterable, but *not* an iterator.

— *weakref_slot* : If true (the default is `False`), add a slot named "__weakref__", which is required to make an instance `weakref-able`. It is an error to specify `weakref_slot=True` without also specifying `slots=True`.

Ajouté dans la version 3.11.

Les champs peuvent éventuellement préciser une valeur par défaut, en utilisant la syntaxe Python normale :

```
@dataclass
class C:
    a: int       # 'a' has no default value
    b: int = 0   # assign a default value for 'b'
```

In this example, both `a` and `b` will be included in the added `__init__()` method, which will be defined as :

```
def __init__(self, a: int, b: int = 0):
```

Une `TypeError` est levée si un champ sans valeur par défaut est défini après un champ avec une valeur par défaut. C'est le cas que ce soit dans une seule classe ou si c'est le résultat d'un héritage de classes.

`dataclasses.field`(*, *default=MISSING*, *default_factory=MISSING*, *init=True*, *repr=True*, *hash=None*, *compare=True*, *metadata=None*, *kw_only=MISSING*, *doc=None*)

For common and simple use cases, no other functionality is required. There are, however, some dataclass features that require additional per-field information. To satisfy this need for additional information, you can replace the default field value with a call to the provided `field()` function. For example :

```
@dataclass
class C:
    mylist: list[int] = field(default_factory=list)

c = C()
c.mylist += [1, 2, 3]
```

Comme le montre la signature, la constante `MISSING` est une valeur sentinelle pour déterminer si des paramètres ont été fournis par l'utilisateur. `None` ne conviendrait pas puisque c'est une valeur avec un sens qui peut être différent pour certains paramètres. La sentinelle `MISSING` est interne au module et ne doit pas être utilisée dans vos programmes.

The parameters to `field()` are :

— *default* : If provided, this will be the default value for this field. This is needed because the `field()` call itself replaces the normal position of the default value.
— *default_factory* : If provided, it must be a zero-argument callable that will be called when a default value is needed for this field. Among other purposes, this can be used to specify fields with mutable default values, as discussed below. It is an error to specify both *default* and *default_factory*.
— *init* : If true (the default), this field is included as a parameter to the generated `__init__()` method.
— *repr* : If true (the default), this field is included in the string returned by the generated `__repr__()` method.
— *hash* : This can be a bool or `None`. If true, this field is included in the generated `__hash__()` method. If `None` (the default), use the value of *compare* : this would normally be the expected behavior. A field should

be considered in the hash if it's used for comparisons. Setting this value to anything other than `None` is discouraged.

Cependant, une raison légitime de mettre *hash* à `False` alors que *compare* est à `True` est la concourance de trois facteurs : le champ est coûteux à hacher ; il est nécessaire pour les comparaisons d'égalité ; et il y a déjà d'autres champs qui participent au hachage des instances. À ce moment, on peut alors se passer du champ dans le hachage tout en le faisant participer aux comparaisons.

— *compare* : If true (the default), this field is included in the generated equality and comparison methods (`__eq__()`, `__gt__()`, et al.).

— *metadata* : This can be a mapping or `None`. `None` is treated as an empty dict. This value is wrapped in `MappingProxyType()` to make it read-only, and exposed on the `Field` object. It is not used at all by Data Classes, and is provided as a third-party extension mechanism. Multiple third-parties can each have their own key, to use as a namespace in the metadata.

— *kw_only* : If true, this field will be marked as keyword-only. This is used when the generated `__init__()` method's parameters are computed.

Ajouté dans la version 3.10.

— `doc` : optional docstring for this field.

Ajouté dans la version 3.13.

If the default value of a field is specified by a call to `field()`, then the class attribute for this field will be replaced by the specified *default* value. If *default* is not provided, then the class attribute will be deleted. The intent is that after the `@dataclass` decorator runs, the class attributes will all contain the default values for the fields, just as if the default value itself were specified. For example, after :

```
@dataclass
class C:
    x: int
    y: int = field(repr=False)
    z: int = field(repr=False, default=10)
    t: int = 20
```

The class attribute `C.z` will be `10`, the class attribute `C.t` will be `20`, and the class attributes `C.x` and `C.y` will not be set.

class dataclasses.**Field**

`Field` objects describe each defined field. These objects are created internally, and are returned by the `fields()` module-level method (see below). Users should never instantiate a `Field` object directly. Its documented attributes are :

— `name` : The name of the field.

— `type` : The type of the field.

— `default`, `default_factory`, `init`, `repr`, `hash`, `compare`, `metadata`, and `kw_only` have the identical meaning and values as they do in the `field()` function.

D'autres attributs peuvent exister, mais ils sont privés et ne sont pas censés être inspectés. Le code ne doit jamais reposer sur eux.

dataclasses.**fields**(*class_or_instance*)

Renvoie un *n*-uplet d'objets `Field` correspondant aux champs de l'argument, à l'exclusion des pseudo-champs `ClassVar` ou `InitVar`. L'argument peut être soit une classe de données, soit une instance d'une telle classe ; si ce n'est pas le cas, une exception `TypeError` est levée.

dataclasses.**asdict**(*obj*, *, *dict_factory=dict*)

Converts the dataclass *obj* to a dict (by using the factory function *dict_factory*). Each dataclass is converted to a dict of its fields, as `name: value` pairs. dataclasses, dicts, lists, and tuples are recursed into. Other objects are copied with `copy.deepcopy()`.

Example of using `asdict()` on nested dataclasses :

```
@dataclass
class Point:
     x: int
     y: int

@dataclass
class C:
     mylist: list[Point]

p = Point(10, 20)
assert asdict(p) == {'x': 10, 'y': 20}

c = C([Point(0, 0), Point(10, 4)])
assert asdict(c) == {'mylist': [{'x': 0, 'y': 0}, {'x': 10, 'y': 4}]}
```

Pour créer une copie superficielle, la solution de contournement suivante peut être utilisée :

```
{field.name: getattr(obj, field.name) for field in fields(obj)}
```

`asdict()` raises `TypeError` if *obj* is not a dataclass instance.

`dataclasses.`**`astuple`**(*obj*, ***, *tuple_factory=tuple*)

Converts the dataclass *obj* to a tuple (by using the factory function *tuple_factory*). Each dataclass is converted to a tuple of its field values. dataclasses, dicts, lists, and tuples are recursed into. Other objects are copied with `copy.deepcopy()`.

Pour continuer l'exemple précédent :

```
assert astuple(p) == (10, 20)
assert astuple(c) == ([(0, 0), (10, 4)],)
```

Pour créer une copie superficielle, la solution de contournement suivante peut être utilisée :

```
tuple(getattr(obj, field.name) for field in dataclasses.fields(obj))
```

`astuple()` raises `TypeError` if *obj* is not a dataclass instance.

`dataclasses.`**`make_dataclass`**(*cls_name*, *fields*, ***, *bases=()*, *namespace=None*, *init=True*, *repr=True*, *eq=True*, *order=False*, *unsafe_hash=False*, *frozen=False*, *match_args=True*, *kw_only=False*, *slots=False*, *weakref_slot=False*, *module=None*, *decorator=dataclass*)

Creates a new dataclass with name *cls_name*, fields as defined in *fields*, base classes as given in *bases*, and initialized with a namespace as given in *namespace*. *fields* is an iterable whose elements are each either `name`, `(name, type)`, or `(name, type, Field)`. If just `name` is supplied, `typing.Any` is used for `type`. The values of *init*, *repr*, *eq*, *order*, *unsafe_hash*, *frozen*, *match_args*, *kw_only*, *slots*, and *weakref_slot* have the same meaning as they do in `@dataclass`.

If *module* is defined, the `__module__` attribute of the dataclass is set to that value. By default, it is set to the module name of the caller.

The *decorator* parameter is a callable that will be used to create the dataclass. It should take the class object as a first argument and the same keyword arguments as `@dataclass`. By default, the `@dataclass` function is used.

This function is not strictly required, because any Python mechanism for creating a new class with `__annotations__` can then apply the `@dataclass` function to convert that class to a dataclass. This function is provided as a convenience. For example :

```
C = make_dataclass('C',
                   [('x', int),
```

(suite sur la page suivante)

(suite de la page précédente)

```
                        'y',
                       ('z', int, field(default=5))],
                      namespace={'add_one': lambda self: self.x + 1})
```

est équivalent à :

```
@dataclass
class C:
    x: int
    y: 'typing.Any'
    z: int = 5

    def add_one(self):
        return self.x + 1
```

Ajouté dans la version 3.14 : Added the *decorator* parameter.

dataclasses.**replace**(*obj*, /, ***changes*)

Creates a new object of the same type as *obj*, replacing fields with values from *changes*. If *obj* is not a Data Class, raises `TypeError`. If keys in *changes* are not field names of the given dataclass, raises `TypeError`.

The newly returned object is created by calling the `__init__()` method of the dataclass. This ensures that `__post_init__()`, if present, is also called.

Init-only variables without default values, if any exist, must be specified on the call to `replace()` so that they can be passed to `__init__()` and `__post_init__()`.

It is an error for *changes* to contain any fields that are defined as having `init=False`. A `ValueError` will be raised in this case.

Be forewarned about how `init=False` fields work during a call to `replace()`. They are not copied from the source object, but rather are initialized in `__post_init__()`, if they're initialized at all. It is expected that `init=False` fields will be rarely and judiciously used. If they are used, it might be wise to have alternate class constructors, or perhaps a custom `replace()` (or similarly named) method which handles instance copying.

Dataclass instances are also supported by generic function `copy.replace()`.

dataclasses.**is_dataclass**(*obj*)

Return `True` if its parameter is a dataclass (including subclasses of a dataclass) or an instance of one, otherwise return `False`.

Pour vérifier qu'un objet *obj* est une instance d'une classe de données, et non pas lui-même une classe de données, ajoutez le test `not isinstance(obj, type)` :

```
def is_dataclass_instance(obj):
    return is_dataclass(obj) and not isinstance(obj, type)
```

dataclasses.**MISSING**

Une valeur sentinelle pour dénoter l'absence de *default* ou *default_factory*.

dataclasses.**KW_ONLY**

A sentinel value used as a type annotation. Any fields after a pseudo-field with the type of `KW_ONLY` are marked as keyword-only fields. Note that a pseudo-field of type `KW_ONLY` is otherwise completely ignored. This includes the name of such a field. By convention, a name of _ is used for a `KW_ONLY` field. Keyword-only fields signify `__init__()` parameters that must be specified as keywords when the class is instantiated.

Dans cet exemple `y` et `z` sont marqués comme exclusivement nommés :

```
@dataclass
class Point:
    x: float
```

(suite sur la page suivante)

(suite de la page précédente)

```
    _: KW_ONLY
    y: float
    z: float

p = Point(0, y=1.5, z=2.0)
```

In a single dataclass, it is an error to specify more than one field whose type is `KW_ONLY`.

Ajouté dans la version 3.10.

exception `dataclasses.`**`FrozenInstanceError`**

Raised when an implicitly defined `__setattr__()` or `__delattr__()` is called on a dataclass which was defined with `frozen=True`. It is a subclass of *`AttributeError`*.

30.7.2 Post-initialisation

`dataclasses.`**`__post_init__`**`()`

When defined on the class, it will be called by the generated `__init__()`, normally as `self.__post_init__()`. However, if any `InitVar` fields are defined, they will also be passed to `__post_init__()` in the order they were defined in the class. If no `__init__()` method is generated, then `__post_init__()` will not automatically be called.

Cette méthode permet, entre autres, d'initialiser des champs qui dépendent d'autres champs. Par exemple :

```
@dataclass
class C:
    a: float
    b: float
    c: float = field(init=False)

    def __post_init__(self):
        self.c = self.a + self.b
```

The `__init__()` method generated by *`@dataclass`* does not call base class `__init__()` methods. If the base class has an `__init__()` method that has to be called, it is common to call this method in a *`__post_init__()`* method :

```
class Rectangle:
    def __init__(self, height, width):
        self.height = height
        self.width = width

@dataclass
class Square(Rectangle):
    side: float

    def __post_init__(self):
        super().__init__(self.side, self.side)
```

Note, however, that in general the dataclass-generated `__init__()` methods don't need to be called, since the derived dataclass will take care of initializing all fields of any base class that is a dataclass itself.

See the section below on init-only variables for ways to pass parameters to `__post_init__()`. Also see the warning about how *`replace()`* handles `init=False` fields.

30.7.3 Variables de classe

One of the few places where *@dataclass* actually inspects the type of a field is to determine if a field is a class variable as defined in **PEP 526**. It does this by checking if the type of the field is *typing.ClassVar*. If a field is a `ClassVar`, it is excluded from consideration as a field and is ignored by the dataclass mechanisms. Such `ClassVar` pseudo-fields are not returned by the module-level *fields()* function.

30.7.4 Variables d'initialisation

Another place where *@dataclass* inspects a type annotation is to determine if a field is an init-only variable. It does this by seeing if the type of a field is of type `dataclasses.InitVar`. If a field is an `InitVar`, it is considered a pseudo-field called an init-only field. As it is not a true field, it is not returned by the module-level *fields()* function. Init-only fields are added as parameters to the generated `__init__()` method, and are passed to the optional *__post_init__()* method. They are not otherwise used by dataclasses.

On peut par exemple imaginer un champ initialisé à partir d'une base de données s'il n'a pas reçu de valeur explicite :

```
@dataclass
class C:
    i: int
    j: int | None = None
    database: InitVar[DatabaseType | None] = None

    def __post_init__(self, database):
        if self.j is None and database is not None:
            self.j = database.lookup('j')

c = C(10, database=my_database)
```

In this case, *fields()* will return *Field* objects for `i` and `j`, but not for `database`.

30.7.5 Instances figées

It is not possible to create truly immutable Python objects. However, by passing `frozen=True` to the *@dataclass* decorator you can emulate immutability. In that case, dataclasses will add `__setattr__()` and `__delattr__()` methods to the class. These methods will raise a *FrozenInstanceError* when invoked.

There is a tiny performance penalty when using `frozen=True` : `__init__()` cannot use simple assignment to initialize fields, and must use `object.__setattr__()`.

30.7.6 Héritage

When the dataclass is being created by the *@dataclass* decorator, it looks through all of the class's base classes in reverse MRO (that is, starting at *object*) and, for each dataclass that it finds, adds the fields from that base class to an ordered mapping of fields. After all of the base class fields are added, it adds its own fields to the ordered mapping. All of the generated methods will use this combined, calculated ordered mapping of fields. Because the fields are in insertion order, derived classes override base classes. An example :

```
@dataclass
class Base:
    x: Any = 15.0
    y: int = 0

@dataclass
class C(Base):
    z: int = 10
    x: int = 15
```

The final list of fields is, in order, `x`, `y`, `z`. The final type of `x` is *int*, as specified in class `C`.

The generated `__init__()` method for `C` will look like :

```
def __init__(self, x: int = 15, y: int = 0, z: int = 10):
```

30.7.7 Re-ordering of keyword-only parameters in `__init__()`

After the parameters needed for `__init__()` are computed, any keyword-only parameters are moved to come after all regular (non-keyword-only) parameters. This is a requirement of how keyword-only parameters are implemented in Python : they must come after non-keyword-only parameters.

In this example, `Base.y`, `Base.w`, and `D.t` are keyword-only fields, and `Base.x` and `D.z` are regular fields :

```
@dataclass
class Base:
    x: Any = 15.0
    _: KW_ONLY
    y: int = 0
    w: int = 1

@dataclass
class D(Base):
    z: int = 10
    t: int = field(kw_only=True, default=0)
```

The generated `__init__()` method for `D` will look like :

```
def __init__(self, x: Any = 15.0, z: int = 10, *, y: int = 0, w: int = 1, t: int =␣
↪0):
```

Les paramètres ont été réarrangés par rapport à leur ordre d'apparition dans la liste des champs : les paramètres provenant des attributs normaux sont suivis par les paramètres qui proviennent des attributs exclusivement nommés.

The relative ordering of keyword-only parameters is maintained in the re-ordered `__init__()` parameter list.

30.7.8 Fabriques de valeurs par défaut

If a *field()* specifies a *default_factory*, it is called with zero arguments when a default value for the field is needed. For example, to create a new instance of a list, use :

```
mylist: list = field(default_factory=list)
```

If a field is excluded from `__init__()` (using `init=False`) and the field also specifies *default_factory*, then the default factory function will always be called from the generated `__init__()` function. This happens because there is no other way to give the field an initial value.

30.7.9 Valeurs par défaut mutables

En Python, les valeurs par défaut des attributs sont stockées dans des attributs de la classe. Observez cet exemple, sans classe de données :

```
class C:
    x = []
    def add(self, element):
        self.x.append(element)

o1 = C()
o2 = C()
o1.add(1)
o2.add(2)
assert o1.x == [1, 2]
assert o1.x is o2.x
```

Note that the two instances of class `C` share the same class variable `x`, as expected.

Avec les classes de données, si ce code était valide :

```
@dataclass
class D:
    x: list = []       # This code raises ValueError
    def add(self, element):
        self.x.append(element)
```

il générerait un code équivalent à :

```
class D:
    x = []
    def __init__(self, x=x):
        self.x = x
    def add(self, element):
        self.x.append(element)

assert D().x is D().x
```

This has the same issue as the original example using class `C`. That is, two instances of class `D` that do not specify a value for `x` when creating a class instance will share the same copy of `x`. Because dataclasses just use normal Python class creation they also share this behavior. There is no general way for Data Classes to detect this condition. Instead, the `@dataclass` decorator will raise a `ValueError` if it detects an unhashable default parameter. The assumption is that if a value is unhashable, it is mutable. This is a partial solution, but it does protect against many common errors.

Pour qu'un champ d'un type mutable soit par défaut initialisé à un nouvel objet pour chaque instance, utilisez une fonction de fabrique :

```
@dataclass
class D:
    x: list = field(default_factory=list)

assert D().x is not D().x
```

Modifié dans la version 3.11 : Instead of looking for and disallowing objects of type `list`, `dict`, or `set`, unhashable objects are now not allowed as default values. Unhashability is used to approximate mutability.

30.7.10 Descriptor-typed fields

Fields that are assigned descriptor objects as their default value have the following special behaviors :

- The value for the field passed to the dataclass's `__init__()` method is passed to the descriptor's `__set__()` method rather than overwriting the descriptor object.
- Similarly, when getting or setting the field, the descriptor's `__get__()` or `__set__()` method is called rather than returning or overwriting the descriptor object.
- To determine whether a field contains a default value, `@dataclass` will call the descriptor's `__get__()` method using its class access form : `descriptor.__get__(obj=None, type=cls)`. If the descriptor returns a value in this case, it will be used as the field's default. On the other hand, if the descriptor raises `AttributeError` in this situation, no default value will be provided for the field.

```
class IntConversionDescriptor:
    def __init__(self, *, default):
        self._default = default

    def __set_name__(self, owner, name):
        self._name = "_" + name
```

(suite sur la page suivante)

(suite de la page précédente)

```
    def __get__(self, obj, type):
        if obj is None:
            return self._default

        return getattr(obj, self._name, self._default)

    def __set__(self, obj, value):
        setattr(obj, self._name, int(value))

@dataclass
class InventoryItem:
    quantity_on_hand: IntConversionDescriptor =␣
↪IntConversionDescriptor(default=100)

i = InventoryItem()
print(i.quantity_on_hand)    # 100
i.quantity_on_hand = 2.5     # calls __set__ with 2.5
print(i.quantity_on_hand)    # 2
```

Note that if a field is annotated with a descriptor type, but is not assigned a descriptor object as its default value, the field will act like a normal field.

30.8 `contextlib` — Utilitaires pour les contextes s'appuyant sur l'instruction `with`

Code source : Lib/contextlib.py

Ce module fournit des utilitaires pour les tâches impliquant le mot-clé `with`. Pour plus d'informations voir aussi *Le type gestionnaire de contexte* et context-managers.

30.8.1 Utilitaires

Fonctions et classes fournies :

class `contextlib.`**`AbstractContextManager`**

Classe mère abstraite pour les classes qui implémentent les méthodes `object.__enter__()` et `object.__exit__()`. Une implémentation par défaut de `object.__enter__()` est fournie, qui renvoie `self`, et `object.__exit__()` est une méthode abstraite qui renvoie `None` par défaut. Voir aussi la définition de *Le type gestionnaire de contexte*.

Ajouté dans la version 3.6.

class `contextlib.`**`AbstractAsyncContextManager`**

Classe mère abstraite pour les classes qui implémentent les méthodes `object.__aenter__()` et `object.__aexit__()`. Une implémentation par défaut de `object.__aenter__()` est fournie, qui renvoie `self`, et `object.__aexit__()` est une méthode abstraite qui renvoie `None` par défaut. Voir aussi la définition de async-context-managers.

Ajouté dans la version 3.7.

`@contextlib.`**`contextmanager`**

This function is a *decorator* that can be used to define a factory function for `with` statement context managers, without needing to create a class or separate `__enter__()` and `__exit__()` methods.

Alors que de nombreux objets s'utilisent nativement dans des blocs *with*, on trouve parfois des ressources qui nécessitent d'être gérées mais ne sont pas des gestionnaires de contextes, et qui n'implémentent pas de méthode `close()` pour pouvoir être utilisées avec `contextlib.closing`

L'exemple abstrait suivant présente comment assurer une gestion correcte des ressources :

```
from contextlib import contextmanager

@contextmanager
def managed_resource(*args, **kwds):
    # Code to acquire resource, e.g.:
    resource = acquire_resource(*args, **kwds)
    try:
        yield resource
    finally:
        # Code to release resource, e.g.:
        release_resource(resource)
```

The function can then be used like this :

```
>>> with managed_resource(timeout=3600) as resource:
...     # Resource is released at the end of this block,
...     # even if code in the block raises an exception
```

La fonction à décorer doit renvoyer un *générateur*-itérateur quand elle est appelée. Ce générateur ne doit produire qu'une seule valeur, qui est récupérée dans le bloc `with` à l'aide de la clause `as` si précisée.

Au moment où le générateur produit une valeur, le bloc imbriqué sous l'instruction `with` est exécuté. Le générateur est ensuite repris après la sortie du bloc. Si une exception non gérée survient dans le bloc, elle est relayée dans le générateur au niveau de l'instruction *yield*. Ainsi, vous pouvez utiliser les instructions `try...except...finally` pour attraper l'erreur (s'il y a), ou vous assurer qu'un nettoyage a bien lieu. Si une exception est attrapée dans l'unique but d'être journalisée ou d'effectuer une action particulière (autre que supprimer entièrement l'exception), le générateur se doit de la relayer. Autrement le générateur gestionnaire de contexte doit indiquer à l'instruction `with` que l'exception a été gérée, et l'exécution reprend sur l'instruction qui suit directement le bloc `with`.

Le décorateur `contextmanager()` utilise la classe `ContextDecorator` afin que les gestionnaires de contexte qu'il crée puissent être utilisés aussi bien en tant que décorateurs qu'avec des instructions `with`. Quand vous l'utilisez comme décorateur, une nouvelle instance du générateur est créée à chaque appel de la fonction (cela permet aux gestionnaires de contexte à usage unique créés par `contextmanager()` de remplir la condition de pouvoir être invoqués plusieurs fois afin d'être utilisés comme décorateurs).

Modifié dans la version 3.2 : Utilisation de la classe `ContextDecorator`.

`@contextlib.`**`asynccontextmanager`**

Similaire à `contextmanager()`, mais crée un gestionnaire de contexte asynchrone.

This function is a *decorator* that can be used to define a factory function for `async with` statement asynchronous context managers, without needing to create a class or separate `__aenter__()` and `__aexit__()` methods. It must be applied to an *asynchronous generator* function.

Un exemple simple :

```
from contextlib import asynccontextmanager

@asynccontextmanager
async def get_connection():
    conn = await acquire_db_connection()
    try:
        yield conn
    finally:
        await release_db_connection(conn)

async def get_all_users():
    async with get_connection() as conn:
```

(suite sur la page suivante)

(suite de la page précédente)

```
        return conn.query('SELECT ...')
```

Ajouté dans la version 3.7.

Les gestionnaires de contexte définis avec *asynccontextmanager()* peuvent s'utiliser comme décorateurs ou dans les instructions `async with` :

```
import time
from contextlib import asynccontextmanager

@asynccontextmanager
async def timeit():
    now = time.monotonic()
    try:
        yield
    finally:
        print(f'it took {time.monotonic() - now}s to run')

@timeit()
async def main():
    # ... async code ...
```

When used as a decorator, a new generator instance is implicitly created on each function call. This allows the otherwise "one-shot" context managers created by *asynccontextmanager()* to meet the requirement that context managers support multiple invocations in order to be used as decorators.

Modifié dans la version 3.10 : Async context managers created with *asynccontextmanager()* can be used as decorators.

`contextlib.`**`closing`**(*thing*)

Renvoie un gestionnaire de contexte qui ferme *thing* à la fin du bloc. C'est essentiellement équivalent à :

```
from contextlib import contextmanager

@contextmanager
def closing(thing):
    try:
        yield thing
    finally:
        thing.close()
```

Et cela vous permet d'écrire du code tel que :

```
from contextlib import closing
from urllib.request import urlopen

with closing(urlopen('https://www.python.org')) as page:
    for line in page:
        print(line)
```

sans besoin de fermer explicitement `page`. Même si une erreur survient, `page.close()` est appelée à la fermeture du bloc `with`.

Note

Most types managing resources support the *context manager* protocol, which closes *thing* on leaving the `with` statement. As such, `closing()` is most useful for third party types that don't support context managers. This example is purely for illustration purposes, as *urlopen()* would normally be used in a context

manager.

contextlib.**aclosing**(*thing*)

Return an async context manager that calls the `aclose()` method of *thing* upon completion of the block. This is basically equivalent to :

```
from contextlib import asynccontextmanager

@asynccontextmanager
async def aclosing(thing):
    try:
        yield thing
    finally:
        await thing.aclose()
```

Significantly, `aclosing()` supports deterministic cleanup of async generators when they happen to exit early by `break` or an exception. For example :

```
from contextlib import aclosing

async with aclosing(my_generator()) as values:
    async for value in values:
        if value == 42:
            break
```

This pattern ensures that the generator's async exit code is executed in the same context as its iterations (so that exceptions and context variables work as expected, and the exit code isn't run after the lifetime of some task it depends on).

Ajouté dans la version 3.10.

contextlib.**nullcontext**(*enter_result=None*)

Renvoie un gestionnaire de contexte dont la méthode `__enter__` renvoie *enter_result*, mais ne fait rien d'autre. L'idée est de l'utiliser comme remplaçant pour un gestionnaire de contexte optionnel, par exemple :

```
def myfunction(arg, ignore_exceptions=False):
    if ignore_exceptions:
        # Use suppress to ignore all exceptions.
        cm = contextlib.suppress(Exception)
    else:
        # Do not ignore any exceptions, cm has no effect.
        cm = contextlib.nullcontext()
    with cm:
        # Do something
```

Un exemple utilisant *enter_result* :

```
def process_file(file_or_path):
    if isinstance(file_or_path, str):
        # If string, open file
        cm = open(file_or_path)
    else:
        # Caller is responsible for closing file
        cm = nullcontext(file_or_path)

    with cm as file:
        # Perform processing on the file
```

It can also be used as a stand-in for asynchronous context managers :

```
async def send_http(session=None):
    if not session:
        # If no http session, create it with aiohttp
        cm = aiohttp.ClientSession()
    else:
        # Caller is responsible for closing the session
        cm = nullcontext(session)

    async with cm as session:
        # Send http requests with session
```

Ajouté dans la version 3.7.

Modifié dans la version 3.10 : *asynchronous context manager* support was added.

`contextlib.`**`suppress`**(**exceptions*)

Return a context manager that suppresses any of the specified exceptions if they occur in the body of a `with` statement and then resumes execution with the first statement following the end of the `with` statement.

Comme pour tous les mécanismes qui suppriment complètement les exceptions, ce gestionnaire de contexte doit seulement être utilisé pour couvrir des cas très spécifiques d'erreurs où il est certain que continuer silencieusement l'exécution du programme est la bonne chose à faire.

Par exemple :

```
from contextlib import suppress

with suppress(FileNotFoundError):
    os.remove('somefile.tmp')

with suppress(FileNotFoundError):
    os.remove('someotherfile.tmp')
```

Ce code est équivalent à :

```
try:
    os.remove('somefile.tmp')
except FileNotFoundError:
    pass

try:
    os.remove('someotherfile.tmp')
except FileNotFoundError:
    pass
```

Ce gestionnaire de contexte est *réentrant*.

If the code within the `with` block raises a `BaseExceptionGroup`, suppressed exceptions are removed from the group. Any exceptions of the group which are not suppressed are re-raised in a new group which is created using the original group's `derive()` method.

Ajouté dans la version 3.4.

Modifié dans la version 3.12 : `suppress` now supports suppressing exceptions raised as part of a `BaseExceptionGroup`.

`contextlib.`**`redirect_stdout`**(*new_target*)

Gestionnaire de contexte servant à rediriger temporairement `sys.stdout` vers un autre fichier ou objet fichier-compatible.

Cet outil ajoute une certaine flexibilité aux fonctions ou classes existantes dont la sortie est envoyée vers la sortie standard.

For example, the output of `help()` normally is sent to *sys.stdout*. You can capture that output in a string by redirecting the output to an `io.StringIO` object. The replacement stream is returned from the `__enter__` method and so is available as the target of the `with` statement :

```
with redirect_stdout(io.StringIO()) as f:
    help(pow)
s = f.getvalue()
```

Pour envoyer la sortie de `help()` vers un fichier sur le disque, redirigez-la sur un fichier normal :

```
with open('help.txt', 'w') as f:
    with redirect_stdout(f):
        help(pow)
```

Pour envoyer la sortie de `help()` sur *sys.stderr* :

```
with redirect_stdout(sys.stderr):
    help(pow)
```

Notez que l'effet de bord global sur `sys.stdout` signifie que ce gestionnaire de contexte n'est pas adapté à une utilisation dans le code d'une bibliothèque ni dans la plupart des applications à plusieurs fils d'exécution. Aussi, cela n'a pas d'effet sur la sortie des sous-processus. Cependant, cela reste une approche utile pour beaucoup de scripts utilitaires.

Ce gestionnaire de contexte est *réentrant*.

Ajouté dans la version 3.4.

`contextlib.`**`redirect_stderr`**(*new_target*)

Similaire à `redirect_stdout()` mais redirige `sys.stderr` vers un autre fichier ou objet fichier-compatible.

Ce gestionnaire de contexte est *réentrant*.

Ajouté dans la version 3.5.

`contextlib.`**`chdir`**(*path*)

Non parallel-safe context manager to change the current working directory. As this changes a global state, the working directory, it is not suitable for use in most threaded or async contexts. It is also not suitable for most non-linear code execution, like generators, where the program execution is temporarily relinquished -- unless explicitly desired, you should not yield when this context manager is active.

This is a simple wrapper around `chdir()`, it changes the current working directory upon entering and restores the old one on exit.

Ce gestionnaire de contexte est *réentrant*.

Ajouté dans la version 3.11.

class `contextlib.`**`ContextDecorator`**

Une classe mère qui permet à un gestionnaire de contexte d'être aussi utilisé comme décorateur.

Les gestionnaires de contexte héritant de `ContextDecorator` doivent implémenter `__enter__` et `__exit__` comme habituellement. `__exit__` conserve sa gestion optionnelle des exceptions même lors de l'utilisation en décorateur.

`ContextDecorator` est utilisé par `contextmanager()`, donc vous bénéficiez automatiquement de cette fonctionnalité.

Exemple de `ContextDecorator` :

```
from contextlib import ContextDecorator

class mycontext(ContextDecorator):
```

(suite sur la page suivante)

(suite de la page précédente)

```
    def __enter__(self):
        print('Starting')
        return self

    def __exit__(self, *exc):
        print('Finishing')
        return False
```

The class can then be used like this :

```
>>> @mycontext()
... def function():
...     print('The bit in the middle')
...
>>> function()
Starting
The bit in the middle
Finishing

>>> with mycontext():
...     print('The bit in the middle')
...
Starting
The bit in the middle
Finishing
```

Ce changement est simplement un sucre syntaxique pour les constructions de la forme suivante :

```
def f():
    with cm():
        # Do stuff
```

`ContextDecorator` vous permet d'écrire à la place :

```
@cm()
def f():
    # Do stuff
```

Cela éclaircit le fait que `cm` s'applique à la fonction entière, et pas seulement à un morceau en particulier (et gagner un niveau d'indentation est toujours appréciable).

Les gestionnaires de contexte existants qui ont déjà une classe mère peuvent être étendus en utilisant `ContextDecorator` comme une *mixin* :

```
from contextlib import ContextDecorator

class mycontext(ContextBaseClass, ContextDecorator):
    def __enter__(self):
        return self

    def __exit__(self, *exc):
        return False
```

Note

Comme la fonction décorée doit être capable d'être appelée plusieurs fois, le gestionnaire de contexte sous-jacent doit permettre d'être utilisé dans de multiples instructions `with`. Si ce n'est pas le cas, alors la

construction d'origine avec de multiples instructions `with` au sein de la fonction doit être utilisée.

Ajouté dans la version 3.2.

class contextlib.**AsyncContextDecorator**

Similar to *ContextDecorator* but only for asynchronous functions.

Example of `AsyncContextDecorator` :

```
from asyncio import run
from contextlib import AsyncContextDecorator

class mycontext(AsyncContextDecorator):
    async def __aenter__(self):
        print('Starting')
        return self

    async def __aexit__(self, *exc):
        print('Finishing')
        return False
```

The class can then be used like this :

```
>>> @mycontext()
... async def function():
...     print('The bit in the middle')
...
>>> run(function())
Starting
The bit in the middle
Finishing

>>> async def function():
...    async with mycontext():
...         print('The bit in the middle')
...
>>> run(function())
Starting
The bit in the middle
Finishing
```

Ajouté dans la version 3.10.

class contextlib.**ExitStack**

Gestionnaire de contexte conçu pour simplifier le fait de combiner programmatiquement d'autres gestionnaires de contexte et fonctions de nettoyage, spécifiquement ceux qui sont optionnels ou pilotés par des données d'entrée.

Par exemple, un ensemble de fichiers peut facilement être géré dans une unique instruction *with* comme suit :

```
with ExitStack() as stack:
    files = [stack.enter_context(open(fname)) for fname in filenames]
    # All opened files will automatically be closed at the end of
    # the with statement, even if attempts to open files later
    # in the list raise an exception
```

The `__enter__()` method returns the *ExitStack* instance, and performs no additional operations.

Chaque instance maintient une pile de fonctions de rappels (*callbacks*) enregistrées qui sont appelées en ordre inverse quand l'instance est fermée (explicitement ou implicitement à la fin d'un bloc `with`). Notez que ces

fonctions ne sont *pas* invoquées implicitement quand l'instance de la pile de contextes est collectée par le ramasse-miettes.

Ce modèle de pile est utilisé afin que les gestionnaires de contexte qui acquièrent leurs ressources dans leur méthode `__init__` (tels que les objets-fichiers) puissent être gérés correctement.

Comme les fonctions de rappel enregistrées sont invoquées dans l'ordre inverse d'enregistrement, cela revient au même que si de multiples blocs `with` imbriqués avaient été utilisés avec l'ensemble de fonctions enregistrées. Cela s'étend aussi à la gestion d'exceptions — si une fonction de rappel intérieure supprime ou remplace une exception, alors les fonctions extérieures reçoivent des arguments basés sur ce nouvel état.

C'est une *API* relativement bas-niveau qui s'occupe de dérouler correctement la pile des appels de sortie. Elle fournit une base adaptée pour des gestionnaires de contexte de plus haut niveau qui manipulent la pile de sortie de manière spécifique à l'application.

Ajouté dans la version 3.3.

enter_context(*cm*)

Enters a new context manager and adds its `__exit__()` method to the callback stack. The return value is the result of the context manager's own `__enter__()` method.

Ces gestionnaires de contexte peuvent supprimer des exceptions comme ils le feraient normalement s'ils étaient utilisés directement derrière une instruction `with`.

Modifié dans la version 3.11 : Raises `TypeError` instead of `AttributeError` if *cm* is not a context manager.

push(*exit*)

Adds a context manager's `__exit__()` method to the callback stack.

As `__enter__` is *not* invoked, this method can be used to cover part of an `__enter__()` implementation with a context manager's own `__exit__()` method.

If passed an object that is not a context manager, this method assumes it is a callback with the same signature as a context manager's `__exit__()` method and adds it directly to the callback stack.

By returning true values, these callbacks can suppress exceptions the same way context manager `__exit__()` methods can.

L'objet passé en paramètre est renvoyé par la fonction, ce qui permet à la méthode d'être utilisée comme décorateur de fonction.

callback(*callback*, /, **args*, ***kwds*)

Accepte une fonction arbitraire et ses arguments et les ajoute à la pile des fonctions de rappel.

À la différence des autres méthodes, les fonctions de rappel ajoutées de cette manière ne peuvent pas supprimer les exceptions (puisqu'elles ne reçoivent jamais les détails de l'exception).

La fonction passée en paramètre est renvoyée par la méthode, ce qui permet à la méthode d'être utilisée comme décorateur de fonction.

pop_all()

Transfère la pile d'appels à une nouvelle instance de `ExitStack` et la renvoie. Aucune fonction de rappel n'est invoquée par cette opération — à la place, elles sont dorénavant invoquées quand la nouvelle pile sera close (soit explicitement soit implicitement à la fin d'un bloc `with`).

Par exemple, un groupe de fichiers peut être ouvert comme une opération « tout ou rien » comme suit :

```
with ExitStack() as stack:
    files = [stack.enter_context(open(fname)) for fname in filenames]
    # Hold onto the close method, but don't call it yet.
    close_files = stack.pop_all().close
    # If opening any file fails, all previously opened files will be
    # closed automatically. If all files are opened successfully,
    # they will remain open even after the with statement ends.
    # close_files() can then be invoked explicitly to close them all.
```

close()

Déroule immédiatement la pile d'appels, invoquant les fonctions de rappel dans l'ordre inverse d'enregistrement. Pour chaque gestionnaire de contexte et fonction de sortie enregistré, les arguments passés indiqueront qu'aucune exception n'est survenue.

class contextlib.**AsyncExitStack**

Un gestionnaire de contexte asynchrone, similaire à *ExitStack*, apte à combiner à la fois des gestionnaires de contexte synchrones et asynchrones, ainsi que la gestion de coroutines pour la logique de nettoyage.

The *close()* method is not implemented ; *aclose()* must be used instead.

coroutine enter_async_context(*cm*)

Similar to *ExitStack.enter_context()* but expects an asynchronous context manager.

Modifié dans la version 3.11 : Raises *TypeError* instead of *AttributeError* if *cm* is not an asynchronous context manager.

push_async_exit(*exit*)

Similar to *ExitStack.push()* but expects either an asynchronous context manager or a coroutine function.

push_async_callback(*callback*, /, **args*, ***kwds*)

Similar to *ExitStack.callback()* but expects a coroutine function.

coroutine aclose()

Similar to *ExitStack.close()* but properly handles awaitables.

En continuité de l'exemple de *asynccontextmanager()* :

```
async with AsyncExitStack() as stack:
    connections = [await stack.enter_async_context(get_connection())
        for i in range(5)]
    # All opened connections will automatically be released at the end of
    # the async with statement, even if attempts to open a connection
    # later in the list raise an exception.
```

Ajouté dans la version 3.7.

30.8.2 Exemples et Recettes

Cette section décrit quelques exemples et recettes pour décrire une utilisation réelle des outils fournis par *contextlib*.

Gérer un nombre variable de gestionnaires de contexte

Le cas d'utilisation primaire de *ExitStack* est celui décrit dans la documentation de la classe : gérer un nombre variable de gestionnaires de contexte et d'autres opérations de nettoyage en une unique instruction `with`. La variabilité peut venir du nombre de gestionnaires de contexte voulus découlant d'une entrée de l'utilisateur (comme ouvrir une collection spécifique de fichiers de l'utilisateur), ou de certains gestionnaires de contexte qui peuvent être optionnels :

```
with ExitStack() as stack:
    for resource in resources:
        stack.enter_context(resource)
    if need_special_resource():
        special = acquire_special_resource()
        stack.callback(release_special_resource, special)
    # Perform operations that use the acquired resources
```

Comme montré, *ExitStack* rend aussi assez facile d'utiliser les instructions `with` pour gérer des ressources arbitraires qui ne gèrent pas nativement le protocole des gestionnaires de contexte.

Attraper des exceptions depuis les méthodes `__enter__`

Il est occasionnellement souhaitable d'attraper les exceptions depuis l'implémentation d'une méthode `__enter__`, *sans* attraper par inadvertance les exceptions du corps de l'instruction `with` ou de la méthode `__exit__` des gestionnaires de contexte. En utilisant *`ExitStack`*, les étapes du protocole des gestionnaires de contexte peuvent être légèrement séparées pour permettre le code suivant :

```
stack = ExitStack()
try:
    x = stack.enter_context(cm)
except Exception:
    # handle __enter__ exception
else:
    with stack:
        # Handle normal case
```

Avoir à faire cela est en fait surtout utile pour indiquer que l'*API* sous-jacente devrait fournir une interface directe de gestion des ressources à utiliser avec les instructions `try`/`except`/`finally`, mais que toutes les *API* ne sont pas bien conçues dans cet objectif. Quand un gestionnaire de contexte est la seule *API* de gestion des ressources fournie, alors *`ExitStack`* peut rendre plus facile la gestion de plusieurs situations qui ne peuvent pas être traitées directement dans une instruction `with`.

Nettoyer dans une méthode `__enter__`

As noted in the documentation of *`ExitStack.push()`*, this method can be useful in cleaning up an already allocated resource if later steps in the `__enter__()` implementation fail.

Voici un exemple de gestionnaire de contexte qui reçoit des fonctions d'acquisition de ressources et de libération, avec une méthode de validation optionnelle, et qui les adapte au protocole des gestionnaires de contexte :

```
from contextlib import contextmanager, AbstractContextManager, ExitStack

class ResourceManager(AbstractContextManager):

    def __init__(self, acquire_resource, release_resource, check_resource_ok=None):
        self.acquire_resource = acquire_resource
        self.release_resource = release_resource
        if check_resource_ok is None:
            def check_resource_ok(resource):
                return True
        self.check_resource_ok = check_resource_ok

    @contextmanager
    def _cleanup_on_error(self):
        with ExitStack() as stack:
            stack.push(self)
            yield
            # The validation check passed and didn't raise an exception
            # Accordingly, we want to keep the resource, and pass it
            # back to our caller
            stack.pop_all()

    def __enter__(self):
        resource = self.acquire_resource()
        with self._cleanup_on_error():
            if not self.check_resource_ok(resource):
                msg = "Failed validation for {!r}"
                raise RuntimeError(msg.format(resource))
        return resource
```

(suite sur la page suivante)

(suite de la page précédente)

```
    def __exit__(self, *exc_details):
        # We don't need to duplicate any of our resource release logic
        self.release_resource()
```

Remplacer un `try-finally` avec une option variable

Un modèle que vous rencontrerez parfois est un bloc `try-finally` avec une option pour indiquer si le corps de la clause `finally` doit être exécuté ou non. Dans sa forme la plus simple (qui ne peut pas déjà être gérée avec juste une clause `except`), cela ressemble à :

```
cleanup_needed = True
try:
    result = perform_operation()
    if result:
        cleanup_needed = False
finally:
    if cleanup_needed:
        cleanup_resources()
```

Comme avec n'importe quel code basé sur une instruction `try`, cela peut poser problème pour le développement et la revue, parce que beaucoup de codes d'installation et de nettoyage peuvent finir par être séparés par des sections de code arbitrairement longues.

`ExitStack` rend possible de plutôt enregistrer une fonction de rappel pour être exécutée à la fin d'une instruction `with`, et décider ensuite de passer l'exécution de cet appel :

```
from contextlib import ExitStack

with ExitStack() as stack:
    stack.callback(cleanup_resources)
    result = perform_operation()
    if result:
        stack.pop_all()
```

This allows the intended cleanup behaviour to be made explicit up front, rather than requiring a separate flag variable.

Si une application particulière utilise beaucoup ce modèle, cela peut-être simplifié encore plus au moyen d'une petite classe d'aide :

```
from contextlib import ExitStack

class Callback(ExitStack):
    def __init__(self, callback, /, *args, **kwds):
        super().__init__()
        self.callback(callback, *args, **kwds)

    def cancel(self):
        self.pop_all()

with Callback(cleanup_resources) as cb:
    result = perform_operation()
    if result:
        cb.cancel()
```

Si le nettoyage de la ressource n'est pas déjà soigneusement embarqué dans une fonction autonome, il est possible d'utiliser le décorateur *`ExitStack.callback()`* pour déclarer la fonction de nettoyage de ressource en avance :

```
from contextlib import ExitStack

with ExitStack() as stack:
    @stack.callback
    def cleanup_resources():
        ...
    result = perform_operation()
    if result:
        stack.pop_all()
```

Dû au fonctionnement du protocole des décorateurs, une fonction déclarée ainsi ne peut prendre aucun paramètre. À la place, les ressources à libérer doivent être récupérées depuis l'extérieur comme des variables de fermeture (*closure*).

Utiliser un gestionnaire de contexte en tant que décorateur de fonction

`ContextDecorator` rend possible l'utilisation d'un gestionnaire de contexte à la fois ordinairement avec une instruction `with` ou comme un décorateur de fonction.

Par exemple, il est parfois utile d'emballer les fonctions ou blocs d'instructions avec un journaliseur qui pourrait suivre le temps d'exécution entre l'entrée et la sortie. Plutôt qu'écrire à la fois un décorateur et un gestionnaire de contexte pour la même tâche, hériter de `ContextDecorator` fournit les deux fonctionnalités en une seule définition :

```
from contextlib import ContextDecorator
import logging

logging.basicConfig(level=logging.INFO)

class track_entry_and_exit(ContextDecorator):
    def __init__(self, name):
        self.name = name

    def __enter__(self):
        logging.info('Entering: %s', self.name)

    def __exit__(self, exc_type, exc, exc_tb):
        logging.info('Exiting: %s', self.name)
```

Les instances de cette classe peuvent être utilisées comme gestionnaires de contexte :

```
with track_entry_and_exit('widget loader'):
    print('Some time consuming activity goes here')
    load_widget()
```

Et comme décorateurs de fonctions :

```
@track_entry_and_exit('widget loader')
def activity():
    print('Some time consuming activity goes here')
    load_widget()
```

Note that there is one additional limitation when using context managers as function decorators : there's no way to access the return value of `__enter__()`. If that value is needed, then it is still necessary to use an explicit `with` statement.

Voir aussi

PEP 343 - The "with" statement
La spécification, les motivations et des exemples de l'instruction `with` en Python.

30.8.3 Gestionnaires de contexte à usage unique, réutilisables et réentrants

La plupart des gestionnaires de contexte sont écrits d'une manière qui ne leur permet que d'être utilisés une fois avec une instruction `with`. Ces gestionnaires de contexte à usage unique doivent être recréés chaque fois qu'ils sont utilisés — tenter de les utiliser une seconde fois lève une exception ou ne fonctionne pas correctement.

Cette limitation commune signifie qu'il est généralement conseillé de créer les gestionnaires de contexte directement dans l'en-tête du bloc `with` où ils sont utilisés (comme montré dans tous les exemples d'utilisation au-dessus).

Les fichiers sont un exemple de gestionnaires de contexte étant effectivement à usage unique, puisque la première instruction `with` ferme le fichier, empêchant d'autres opérations d'entrée/sortie d'être exécutées sur ce fichier.

Les gestionnaires de contexte créés avec *`contextmanager()`* sont aussi à usage unique, et se plaindront du fait que le générateur sous-jacent ne produise plus de valeur si vous essayez de les utiliser une seconde fois :

```
>>> from contextlib import contextmanager
>>> @contextmanager
... def singleuse():
...     print("Before")
...     yield
...     print("After")
...
>>> cm = singleuse()
>>> with cm:
...     pass
...
Before
After
>>> with cm:
...     pass
...
Traceback (most recent call last):
    ...
RuntimeError: generator didn't yield
```

Gestionnaires de contexte réentrants

Certains gestionnaires de contexte plus sophistiqués peuvent être « réentrants ». Ces gestionnaires de contexte ne peuvent pas seulement être utilisés avec plusieurs instructions `with`, mais aussi *à l'intérieur* d'une instruction `with` qui utilise déjà ce même gestionnaire de contexte.

`threading.RLock` is an example of a reentrant context manager, as are *`suppress()`*, *`redirect_stdout()`*, and *`chdir()`*. Here's a very simple example of reentrant use :

```
>>> from contextlib import redirect_stdout
>>> from io import StringIO
>>> stream = StringIO()
>>> write_to_stream = redirect_stdout(stream)
>>> with write_to_stream:
...     print("This is written to the stream rather than stdout")
...     with write_to_stream:
...         print("This is also written to the stream")
...
>>> print("This is written directly to stdout")
This is written directly to stdout
>>> print(stream.getvalue())
This is written to the stream rather than stdout
This is also written to the stream
```

Les exemples plus réels de réentrance sont susceptibles d'invoquer plusieurs fonctions s'entre-appelant, et donc être bien plus compliqués que cet exemple.

Notez aussi qu'être réentrant ne signifie *pas* être *thread safe*. `redirect_stdout()`, par exemple, n'est définitivement pas *thread safe*, puisqu'il effectue des changements globaux sur l'état du système en branchant `sys.stdout` sur différents flux.

Gestionnaires de contexte réutilisables

D'autres gestionnaires de contexte que ceux à usage unique et les réentrants sont les gestionnaires de contexte « réutilisables » (ou, pour être plus explicite, « réutilisables mais pas réentrants », puisque les gestionnaires de contexte réentrants sont aussi réutilisables). Ces gestionnaires de contexte sont conçus afin d'être utilisés plusieurs fois, mais échoueront (ou ne fonctionnent pas correctement) si l'instance de gestionnaire de contexte référencée a déjà été utilisée dans une instruction *with* englobante.

`threading.Lock` est un exemple de gestionnaire de contexte réutilisable mais pas réentrant (pour un verrou réentrant, il faut à la place utiliser `threading.RLock`).

Un autre exemple de gestionnaire de contexte réutilisable mais pas réentrant est `ExitStack`, puisqu'il invoque *toutes* les fonctions de rappel actuellement enregistrées en quittant l'instruction *with*, sans regarder où ces fonctions ont été ajoutées :

```
>>> from contextlib import ExitStack
>>> stack = ExitStack()
>>> with stack:
...     stack.callback(print, "Callback: from first context")
...     print("Leaving first context")
...
Leaving first context
Callback: from first context
>>> with stack:
...     stack.callback(print, "Callback: from second context")
...     print("Leaving second context")
...
Leaving second context
Callback: from second context
>>> with stack:
...     stack.callback(print, "Callback: from outer context")
...     with stack:
...         stack.callback(print, "Callback: from inner context")
...         print("Leaving inner context")
...     print("Leaving outer context")
...
Leaving inner context
Callback: from inner context
Callback: from outer context
Leaving outer context
```

Comme le montre la sortie de l'exemple, réutiliser une simple pile entre plusieurs instructions *with* fonctionne correctement, mais essayer de les imbriquer fait que la pile est vidée à la fin du *with* le plus imbriqué, ce qui n'est probablement pas le comportement voulu.

Pour éviter ce problème, utilisez des instances différentes de `ExitStack` plutôt qu'une seule instance :

```
>>> from contextlib import ExitStack
>>> with ExitStack() as outer_stack:
...     outer_stack.callback(print, "Callback: from outer context")
...     with ExitStack() as inner_stack:
...         inner_stack.callback(print, "Callback: from inner context")
...         print("Leaving inner context")
...     print("Leaving outer context")
...
```

(suite sur la page suivante)

(suite de la page précédente)

```
Leaving inner context
Callback: from inner context
Leaving outer context
Callback: from outer context
```

30.9 abc --- Abstract Base Classes

Code source : Lib/abc.py

Le module fournit l'infrastructure pour définir les *classes mères abstraites* (*Abstract Base Class* ou *ABC* en anglais) en Python, tel qu'indiqué dans la **PEP 3119** ; voir la PEP pour la raison de son ajout à Python. (Voir également la **PEP 3141** et le module `numbers` pour ce qui concerne la hiérarchie de types pour les nombres basés sur les classes mères abstraites). Par la suite nous utiliserons l'abréviation ABC (*Abstract Base Class*) pour désigner une classe mère abstraite.

The `collections` module has some concrete classes that derive from ABCs ; these can, of course, be further derived. In addition, the `collections.abc` submodule has some ABCs that can be used to test whether a class or instance provides a particular interface, for example, if it is *hashable* or if it is a *mapping*.

Ce module fournit la métaclasse `ABCMeta` pour définir les ABC ainsi que la classe d'aide `ABC`, cette dernière permettant de définir des ABC en utilisant l'héritage :

class abc.**ABC**

A helper class that has `ABCMeta` as its metaclass. With this class, an abstract base class can be created by simply deriving from `ABC` avoiding sometimes confusing metaclass usage, for example :

```
from abc import ABC

class MyABC(ABC):
    pass
```

Note that the type of `ABC` is still `ABCMeta`, therefore inheriting from `ABC` requires the usual precautions regarding metaclass usage, as multiple inheritance may lead to metaclass conflicts. One may also define an abstract base class by passing the metaclass keyword and using `ABCMeta` directly, for example :

```
from abc import ABCMeta

class MyABC(metaclass=ABCMeta):
    pass
```

Ajouté dans la version 3.4.

class abc.**ABCMeta**

Métaclasse pour définir des classes mères abstraites (ABC).

Utilisez cette métaclasse pour créer une ABC. Il est possible d'hériter d'une ABC directement, cette classe mère abstraite fonctionne alors comme une classe *mixin*. Vous pouvez également enregistrer une classe concrète sans lien (même une classe native) et des ABC comme "sous-classes virtuelles" -- celles-ci et leur descendantes seront considérées comme des sous-classes de la classe mère abstraite par la fonction native `issubclass()`, mais les ABC enregistrées n'apparaitront pas dans leur ordre de résolution des méthodes (*MRO* pour *Method Resolution Order* en anglais). Les implémentations de méthodes définies par l'ABC ne seront pas appelable (pas même via `super()`). [1]

Classes created with a metaclass of `ABCMeta` have the following method :

1. Les développeurs C++ noteront que le concept Python de classe mère virtuelle (*virtual base class*) n'est pas le même que celui de C++.

register(*subclass*)

Enregistrer *subclass* en tant que sous-classe virtuelle de cette ABC. Par exemple :

```
from abc import ABC

class MyABC(ABC):
    pass

MyABC.register(tuple)

assert issubclass(tuple, MyABC)
assert isinstance((), MyABC)
```

Modifié dans la version 3.3 : Renvoie la sous-classe enregistrée pour permettre l'utilisation en tant que décorateur de classe.

Modifié dans la version 3.4 : To detect calls to `register()`, you can use the *get_cache_token()* function.

Vous pouvez également redéfinir cette méthode dans une ABC :

__subclasshook__(*subclass*)

(Doit être définie en tant que méthode de classe.)

Check whether *subclass* is considered a subclass of this ABC. This means that you can customize the behavior of *issubclass()* further without the need to call *register()* on every class you want to consider a subclass of the ABC. (This class method is called from the `__subclasscheck__()` method of the ABC.)

This method should return `True`, `False` or *NotImplemented*. If it returns `True`, the *subclass* is considered a subclass of this ABC. If it returns `False`, the *subclass* is not considered a subclass of this ABC, even if it would normally be one. If it returns `NotImplemented`, the subclass check is continued with the usual mechanism.

Pour une illustration de ces concepts, voir cet exemple de définition de ABC :

```
class Foo:
    def __getitem__(self, index):
        ...
    def __len__(self):
        ...
    def get_iterator(self):
        return iter(self)

class MyIterable(ABC):

    @abstractmethod
    def __iter__(self):
        while False:
            yield None

    def get_iterator(self):
        return self.__iter__()

    @classmethod
    def __subclasshook__(cls, C):
        if cls is MyIterable:
            if any("__iter__" in B.__dict__ for B in C.__mro__):
                return True
        return NotImplemented
```

(suite sur la page suivante)

(suite de la page précédente)

```
MyIterable.register(Foo)
```

The ABC `MyIterable` defines the standard iterable method, *__iter__()*, as an abstract method. The implementation given here can still be called from subclasses. The `get_iterator()` method is also part of the `MyIterable` abstract base class, but it does not have to be overridden in non-abstract derived classes.

The *__subclasshook__()* class method defined here says that any class that has an *__iter__()* method in its `__dict__` (or in that of one of its base classes, accessed via the `__mro__` list) is considered a `MyIterable` too.

Finally, the last line makes `Foo` a virtual subclass of `MyIterable`, even though it does not define an *__iter__()* method (it uses the old-style iterable protocol, defined in terms of `__len__()` and `__getitem__()`). Note that this will not make `get_iterator` available as a method of `Foo`, so it is provided separately.

The `abc` module also provides the following decorator :

`@abc.`**`abstractmethod`**

Un décorateur marquant les méthodes comme abstraites.

Using this decorator requires that the class's metaclass is *ABCMeta* or is derived from it. A class that has a metaclass derived from `ABCMeta` cannot be instantiated unless all of its abstract methods and properties are overridden. The abstract methods can be called using any of the normal 'super' call mechanisms. `abstractmethod()` may be used to declare abstract methods for properties and descriptors.

Dynamically adding abstract methods to a class, or attempting to modify the abstraction status of a method or class once it is created, are only supported using the *update_abstractmethods()* function. The `abstractmethod()` only affects subclasses derived using regular inheritance ; "virtual subclasses" registered with the ABC's *register()* method are not affected.

When `abstractmethod()` is applied in combination with other method descriptors, it should be applied as the innermost decorator, as shown in the following usage examples :

```
class C(ABC):
    @abstractmethod
    def my_abstract_method(self, arg1):
        ...
    @classmethod
    @abstractmethod
    def my_abstract_classmethod(cls, arg2):
        ...
    @staticmethod
    @abstractmethod
    def my_abstract_staticmethod(arg3):
        ...

    @property
    @abstractmethod
    def my_abstract_property(self):
        ...
    @my_abstract_property.setter
    @abstractmethod
    def my_abstract_property(self, val):
        ...

    @abstractmethod
    def _get_x(self):
        ...
    @abstractmethod
```

(suite sur la page suivante)

(suite de la page précédente)

```
    def _set_x(self, val):
        ...
    x = property(_get_x, _set_x)
```

In order to correctly interoperate with the abstract base class machinery, the descriptor must identify itself as abstract using `__isabstractmethod__`. In general, this attribute should be `True` if any of the methods used to compose the descriptor are abstract. For example, Python's built-in *property* does the equivalent of :

```
class Descriptor:
    ...
    @property
    def __isabstractmethod__(self):
        return any(getattr(f, '__isabstractmethod__', False) for
                   f in (self._fget, self._fset, self._fdel))
```

> **Note**
>
> Contrairement aux méthodes abstraites Java, ces méthodes abstraites peuvent être implémentées. Cette implémentation peut être appelée via le mécanisme *super()* depuis la classe qui la redéfinit. C'est typiquement utile pour y appeler *super* et ainsi coopérer correctement dans un environnement utilisant de l'héritage multiple.

The `abc` module also supports the following legacy decorators :

`@abc.`**`abstractclassmethod`**

Ajouté dans la version 3.2.

Obsolète depuis la version 3.3 : Il est désormais possible d'utiliser *classmethod* avec *abstractmethod()*, cela rend ce décorateur redondant.

Sous-classe du décorateur natif *classmethod()* qui indique une méthode de classe (`classmethod`) abstraite. En dehors de cela, est similaire à *abstractmethod()*.

Ce cas spécial est obsolète car le décorateur *classmethod()* est désormais correctement identifié comme abstrait quand il est appliqué à une méthode abstraite :

```
class C(ABC):
    @classmethod
    @abstractmethod
    def my_abstract_classmethod(cls, arg):
        ...
```

`@abc.`**`abstractstaticmethod`**

Ajouté dans la version 3.2.

Obsolète depuis la version 3.3 : Il est désormais possible d'utiliser *staticmethod* avec *abstractmethod()*, cela rend ce décorateur redondant.

Sous-classe du décorateur natif *classmethod()* qui indique une méthode statique (`staticmethod`) abstraite. En dehors de cela, est similaire à *abstractmethod()*.

Ce cas spécial est obsolète car le décorateur *staticmethod()* est désormais correctement identifié comme abstrait quand appliqué à une méthode abstraite :

```
class C(ABC):
    @staticmethod
    @abstractmethod
    def my_abstract_staticmethod(arg):
        ...
```

`@abc.abstractproperty`

Obsolète depuis la version 3.3 : Il est désormais possible d'utiliser `property`, `property.getter()`, `property.setter()` et `property.deleter()` avec `abstractmethod()`, ce qui rend ce décorateur redondant.

Sous-classe de `property()`, qui indique une propriété abstraite.

Ce cas spécial est obsolète car le décorateur `property()` est désormais correctement identifié comme abstrait quand appliqué à une méthode abstraite :

```
class C(ABC):
    @property
    @abstractmethod
    def my_abstract_property(self):
        ...
```

L'exemple ci-dessus définit une propriété en lecture seule. Vous pouvez également définir une propriété en lecture-écriture abstraite en indiquant une ou plusieurs des méthodes sous-jacentes comme abstraite :

```
class C(ABC):
    @property
    def x(self):
        ...

    @x.setter
    @abstractmethod
    def x(self, val):
        ...
```

Si seuls certains composants sont abstraits, seuls ces composants abstraits nécessitent d'être mis à jour pour créer une propriété concrète dans une sous-classe :

```
class D(C):
    @C.x.setter
    def x(self, val):
        ...
```

The `abc` module also provides the following functions :

`abc.get_cache_token()`

Renvoie le jeton de cache (*cache token*) de l'ABC.

Le jeton est un objet opaque (qui implémente le test d'égalité) qui identifie la version actuelle du cache de l'ABC pour les sous-classes virtuelles. Le jeton change avec chaque appel à `ABCMeta.register()` sur n'importe quelle ABC.

Ajouté dans la version 3.4.

`abc.update_abstractmethods(cls)`

Recalcule l'état d'abstraction de la classe. Il est nécessaire d'appeler cette fonction si les méthodes abstraites d'une classe sont ajoutées ou modifiées après la création de la classe. C'est notamment le cas dans les décorateurs de classe.

Pour permettre une utilisation en tant que décorateur, cette fonction renvoie *cls*.

Ne fait rien si *cls* n'est pas une instance de `ABCMeta`.

> **Note**
>
> Cette fonction suppose que les classes mères de *cls* ont vu leur état recalculé, et ne fait rien pour recalculer celui des classes filles.

Ajouté dans la version 3.10.

Notes

30.10 atexit --- Exit handlers

Le module atexit définit des fonctions pour inscrire et désinscrire des fonctions de nettoyage. Les fonctions ainsi inscrites sont automatiquement exécutées au moment de l'arrêt normal de l'interpréteur. atexit exécute ces fonctions dans l'ordre inverse dans lequel elles ont été inscrites ; si vous inscrivez A, B, et C, au moment de l'arrêt de l'interpréteur elles seront exécutées dans l'ordre C, B, A.

Note : Les fonctions inscrites via ce module ne sont pas appelées quand le programme est tué par un signal non géré par Python, quand une erreur fatale interne de Python est détectée, ou quand os._exit() est appelé.

Note : The effect of registering or unregistering functions from within a cleanup function is undefined.

Modifié dans la version 3.7 : Quand elles sont utilisées avec des sous-interpréteurs de l'API C, les fonctions inscrites sont locales à l'interpréteur dans lequel elles ont été inscrites.

atexit.**register**(*func*, **args*, ***kwargs*)

Inscrit *func* comme une fonction à exécuter au moment de l'arrêt de l'interpréteur. Tout argument optionnel qui doit être passé à *func* doit être passé comme argument à register(). Il est possible d'inscrire les mêmes fonctions et arguments plus d'une fois.

Lors d'un arrêt normal du programme (par exemple, si sys.exit() est appelée ou l'exécution du module principal se termine), toutes les fonctions inscrites sont appelées, dans l'ordre de la dernière arrivée, première servie. La supposition est que les modules les plus bas niveau vont normalement être importés avant les modules haut niveau et ainsi être nettoyés en dernier.

Si une exception est levée durant l'exécution du gestionnaire de fin de programme, une trace d'appels est affichée (à moins que SystemExit ait été levée) et les informations de l'exception sont sauvegardées. Une fois que tous les gestionnaires de fin de programme ont eu une chance de s'exécuter, la dernière exception à avoir été levée l'est de nouveau.

Cette fonction renvoie *func*, ce qui rend possible de l'utiliser en tant que décorateur.

> **Avertissement**
>
> Starting new threads or calling os.fork() from a registered function can lead to race condition between the main Python runtime thread freeing thread states while internal threading routines or the new process try to use that state. This can lead to crashes rather than clean shutdown.

Modifié dans la version 3.12 : Attempts to start a new thread or os.fork() a new process in a registered function now leads to RuntimeError.

atexit.**unregister**(*func*)

Retire *func* de la liste des fonctions à exécuter au moment de l'arrêt de l'interpréteur. unregister() ne fait rien et reste silencieux si *func* n'a pas été préalablement inscrite. Si *func* a été inscrite plus d'une fois, toutes les occurrences de cette fonction sont retirées de la pile d'appels de atexit. La comparaison d'égalité (==) est utilisée dans l'implémentation interne de la désinscription. Les références des fonctions n'ont donc pas besoin d'avoir la même identité.

> **Voir aussi**
>
> **Module readline**
> Un exemple utile de l'usage de atexit pour lire et écrire des fichiers d'historique readline.

30.10.1 Exemple avec `atexit`

Le simple exemple suivant démontre comment un module peut initialiser un compteur depuis un fichier quand il est importé, et sauver le valeur mise à jour du compteur automatiquement quand le programme se termine, sans avoir besoin que l'application fasse un appel explicite dans ce module au moment de l'arrêt de l'interpréteur.

```
try:
    with open('counterfile') as infile:
        _count = int(infile.read())
except FileNotFoundError:
    _count = 0

def incrcounter(n):
    global _count
    _count = _count + n

def savecounter():
    with open('counterfile', 'w') as outfile:
        outfile.write('%d' % _count)

import atexit

atexit.register(savecounter)
```

Les arguments positionnels et nommés peuvent aussi être passés à `register()` afin d'être repassés à la fonction inscrite lors de son appel :

```
def goodbye(name, adjective):
    print('Goodbye %s, it was %s to meet you.' % (name, adjective))

import atexit

atexit.register(goodbye, 'Donny', 'nice')
# or:
atexit.register(goodbye, adjective='nice', name='Donny')
```

Utilisation en tant que *décorateur* :

```
import atexit

@atexit.register
def goodbye():
    print('You are now leaving the Python sector.')
```

Ceci fonctionne uniquement avec des fonctions qui peuvent être appelées sans argument.

30.11 `traceback` --- Print or retrieve a stack traceback

Source code : Lib/traceback.py

This module provides a standard interface to extract, format and print stack traces of Python programs. It is more flexible than the interpreter's default traceback display, and therefore makes it possible to configure certain aspects of the output. Finally, it contains a utility for capturing enough information about an exception to print it later, without the need to save a reference to the actual exception. Since exceptions can be the roots of large objects graph, this utility can significantly improve memory management.

The module uses traceback objects --- these are objects of type `types.TracebackType`, which are assigned to the `__traceback__` field of `BaseException` instances.

Voir aussi

Module *faulthandler*
Used to dump Python tracebacks explicitly, on a fault, after a timeout, or on a user signal.

Module *pdb*
Interactive source code debugger for Python programs.

The module's API can be divided into two parts :

— Module-level functions offering basic functionality, which are useful for interactive inspection of exceptions and tracebacks.

— `TracebackException` class and its helper classes `StackSummary` and `FrameSummary`. These offer both more flexibility in the output generated and the ability to store the information necessary for later formatting without holding references to actual exception and traceback objects.

Ajouté dans la version 3.13 : Output is colorized by default and can be controlled using environment variables.

30.11.1 Fonctions de niveau module

`traceback.`**`print_tb`**(*tb*, *limit=None*, *file=None*)

Print up to *limit* stack trace entries from traceback object *tb* (starting from the caller's frame) if *limit* is positive. Otherwise, print the last `abs(limit)` entries. If *limit* is omitted or `None`, all entries are printed. If *file* is omitted or `None`, the output goes to `sys.stderr`; otherwise it should be an open *file* or *file-like object* to receive the output.

Note

The meaning of the *limit* parameter is different than the meaning of `sys.tracebacklimit`. A negative *limit* value corresponds to a positive value of `sys.tracebacklimit`, whereas the behaviour of a positive *limit* value cannot be achieved with `sys.tracebacklimit`.

Modifié dans la version 3.5 : Added negative *limit* support.

`traceback.`**`print_exception`**(*exc*, /, [*value*, *tb*,]*limit=None*, *file=None*, *chain=True*)

Print exception information and stack trace entries from traceback object *tb* to *file*. This differs from `print_tb()` in the following ways :

— if *tb* is not `None`, it prints a header `Traceback (most recent call last):`

— it prints the exception type and *value* after the stack trace

— if *type(value)* is `SyntaxError` and *value* has the appropriate format, it prints the line where the syntax error occurred with a caret indicating the approximate position of the error.

Since Python 3.10, instead of passing *value* and *tb*, an exception object can be passed as the first argument. If *value* and *tb* are provided, the first argument is ignored in order to provide backwards compatibility.

The optional *limit* argument has the same meaning as for `print_tb()`. If *chain* is true (the default), then chained exceptions (the `__cause__` or `__context__` attributes of the exception) will be printed as well, like the interpreter itself does when printing an unhandled exception.

Modifié dans la version 3.5 : The *etype* argument is ignored and inferred from the type of *value*.

Modifié dans la version 3.10 : The *etype* parameter has been renamed to *exc* and is now positional-only.

`traceback.`**`print_exc`**(*limit=None*, *file=None*, *chain=True*)

This is a shorthand for `print_exception(sys.exception(), limit, file, chain)`.

`traceback.`**`print_last`**(*limit=None*, *file=None*, *chain=True*)

This is a shorthand for `print_exception(sys.last_exc, limit, file, chain)`. In general it will work only after an exception has reached an interactive prompt (see `sys.last_exc`).

traceback.**print_stack**(*f=None*, *limit=None*, *file=None*)

Print up to *limit* stack trace entries (starting from the invocation point) if *limit* is positive. Otherwise, print the last `abs(limit)` entries. If *limit* is omitted or `None`, all entries are printed. The optional *f* argument can be used to specify an alternate stack frame to start. The optional *file* argument has the same meaning as for `print_tb()`.

Modifié dans la version 3.5 : Added negative *limit* support.

traceback.**extract_tb**(*tb*, *limit=None*)

Return a `StackSummary` object representing a list of "pre-processed" stack trace entries extracted from the traceback object *tb*. It is useful for alternate formatting of stack traces. The optional *limit* argument has the same meaning as for `print_tb()`. A "pre-processed" stack trace entry is a `FrameSummary` object containing attributes `filename`, `lineno`, `name`, and `line` representing the information that is usually printed for a stack trace.

traceback.**extract_stack**(*f=None*, *limit=None*)

Extract the raw traceback from the current stack frame. The return value has the same format as for `extract_tb()`. The optional *f* and *limit* arguments have the same meaning as for `print_stack()`.

traceback.**print_list**(*extracted_list*, *file=None*)

Print the list of tuples as returned by `extract_tb()` or `extract_stack()` as a formatted stack trace to the given file. If *file* is `None`, the output is written to `sys.stderr`.

traceback.**format_list**(*extracted_list*)

Given a list of tuples or `FrameSummary` objects as returned by `extract_tb()` or `extract_stack()`, return a list of strings ready for printing. Each string in the resulting list corresponds to the item with the same index in the argument list. Each string ends in a newline ; the strings may contain internal newlines as well, for those items whose source text line is not `None`.

traceback.**format_exception_only**(*exc*, /, [*value*,]*, *show_group=False*)

Format the exception part of a traceback using an exception value such as given by `sys.last_value`. The return value is a list of strings, each ending in a newline. The list contains the exception's message, which is normally a single string ; however, for `SyntaxError` exceptions, it contains several lines that (when printed) display detailed information about where the syntax error occurred. Following the message, the list contains the exception's `notes`.

Since Python 3.10, instead of passing *value*, an exception object can be passed as the first argument. If *value* is provided, the first argument is ignored in order to provide backwards compatibility.

When *show_group* is `True`, and the exception is an instance of `BaseExceptionGroup`, the nested exceptions are included as well, recursively, with indentation relative to their nesting depth.

Modifié dans la version 3.10 : The *etype* parameter has been renamed to *exc* and is now positional-only.

Modifié dans la version 3.11 : The returned list now includes any `notes` attached to the exception.

Modifié dans la version 3.13 : *show_group* parameter was added.

traceback.**format_exception**(*exc*, /, [*value*, *tb*,]*limit=None*, *chain=True*)

Format a stack trace and the exception information. The arguments have the same meaning as the corresponding arguments to `print_exception()`. The return value is a list of strings, each ending in a newline and some containing internal newlines. When these lines are concatenated and printed, exactly the same text is printed as does `print_exception()`.

Modifié dans la version 3.5 : The *etype* argument is ignored and inferred from the type of *value*.

Modifié dans la version 3.10 : This function's behavior and signature were modified to match `print_exception()`.

traceback.**format_exc**(*limit=None*, *chain=True*)

This is like `print_exc(limit)` but returns a string instead of printing to a file.

traceback.**format_tb**(*tb*, *limit=None*)

A shorthand for `format_list(extract_tb(tb, limit))`.

traceback.**format_stack**(*f=None*, *limit=None*)

A shorthand for `format_list(extract_stack(f, limit))`.

traceback.**clear_frames**(*tb*)

Clears the local variables of all the stack frames in a traceback *tb* by calling the `clear()` method of each frame object.

Ajouté dans la version 3.4.

traceback.**walk_stack**(*f*)

Walk a stack following `f.f_back` from the given frame, yielding the frame and line number for each frame. If *f* is `None`, the current stack is used. This helper is used with `StackSummary.extract()`.

Ajouté dans la version 3.5.

Modifié dans la version 3.14 : This function previously returned a generator that would walk the stack when first iterated over. The generator returned now is the state of the stack when `walk_stack` is called.

traceback.**walk_tb**(*tb*)

Walk a traceback following `tb_next` yielding the frame and line number for each frame. This helper is used with `StackSummary.extract()`.

Ajouté dans la version 3.5.

30.11.2 `TracebackException` Objects

Ajouté dans la version 3.5.

`TracebackException` objects are created from actual exceptions to capture data for later printing. They offer a more lightweight method of storing this information by avoiding holding references to traceback and frame objects. In addition, they expose more options to configure the output compared to the module-level functions described above.

class traceback.**TracebackException**(*exc_type*, *exc_value*, *exc_traceback*, ***, *limit=None*, *lookup_lines=True*, *capture_locals=False*, *compact=False*, *max_group_width=15*, *max_group_depth=10*)

Capture an exception for later rendering. The meaning of *limit*, *lookup_lines* and *capture_locals* are as for the `StackSummary` class.

If *compact* is true, only data that is required by `TracebackException`'s `format()` method is saved in the class attributes. In particular, the `__context__` field is calculated only if `__cause__` is `None` and `__suppress_context__` is false.

Note that when locals are captured, they are also shown in the traceback.

max_group_width and *max_group_depth* control the formatting of exception groups (see `BaseExceptionGroup`). The depth refers to the nesting level of the group, and the width refers to the size of a single exception group's exceptions array. The formatted output is truncated when either limit is exceeded.

Modifié dans la version 3.10 : Ajout du paramètre *compact*.

Modifié dans la version 3.11 : Added the *max_group_width* and *max_group_depth* parameters.

__cause__

A `TracebackException` of the original `__cause__`.

__context__

A `TracebackException` of the original `__context__`.

exceptions

If `self` represents an `ExceptionGroup`, this field holds a list of `TracebackException` instances representing the nested exceptions. Otherwise it is `None`.

Ajouté dans la version 3.11.

__suppress_context__

The *__suppress_context__* value from the original exception.

__notes__

The *__notes__* value from the original exception, or `None` if the exception does not have any notes. If it is not `None` is it formatted in the traceback after the exception string.

Ajouté dans la version 3.11.

stack

A *StackSummary* representing the traceback.

exc_type

The class of the original traceback.

Obsolète depuis la version 3.13.

exc_type_str

String display of the class of the original exception.

Ajouté dans la version 3.13.

filename

For syntax errors - the file name where the error occurred.

lineno

For syntax errors - the line number where the error occurred.

end_lineno

For syntax errors - the end line number where the error occurred. Can be `None` if not present.

Ajouté dans la version 3.10.

text

For syntax errors - the text where the error occurred.

offset

For syntax errors - the offset into the text where the error occurred.

end_offset

For syntax errors - the end offset into the text where the error occurred. Can be `None` if not present.

Ajouté dans la version 3.10.

msg

For syntax errors - the compiler error message.

classmethod from_exception(*exc*, ***, *limit=None*, *lookup_lines=True*, *capture_locals=False*)

Capture an exception for later rendering. *limit*, *lookup_lines* and *capture_locals* are as for the *StackSummary* class.

Note that when locals are captured, they are also shown in the traceback.

print(***, *file=None*, *chain=True*)

Print to *file* (default `sys.stderr`) the exception information returned by *format()*.

Ajouté dans la version 3.11.

format(***, *chain=True*)

Format the exception.

If *chain* is not `True`, *__cause__* and *__context__* will not be formatted.

The return value is a generator of strings, each ending in a newline and some containing internal newlines. *print_exception()* is a wrapper around this method which just prints the lines to a file.

format_exception_only(*, *show_group=False*)

Format the exception part of the traceback.

The return value is a generator of strings, each ending in a newline.

When *show_group* is `False`, the generator emits the exception's message followed by its notes (if it has any). The exception message is normally a single string; however, for `SyntaxError` exceptions, it consists of several lines that (when printed) display detailed information about where the syntax error occurred.

When *show_group* is `True`, and the exception is an instance of `BaseExceptionGroup`, the nested exceptions are included as well, recursively, with indentation relative to their nesting depth.

Modifié dans la version 3.11 : The exception's `notes` are now included in the output.

Modifié dans la version 3.13 : Added the *show_group* parameter.

30.11.3 `StackSummary` Objects

Ajouté dans la version 3.5.

`StackSummary` objects represent a call stack ready for formatting.

class traceback.**StackSummary**

classmethod extract(*frame_gen*, *, *limit=None*, *lookup_lines=True*, *capture_locals=False*)

Construct a `StackSummary` object from a frame generator (such as is returned by `walk_stack()` or `walk_tb()`).

If *limit* is supplied, only this many frames are taken from *frame_gen*. If *lookup_lines* is `False`, the returned `FrameSummary` objects will not have read their lines in yet, making the cost of creating the `StackSummary` cheaper (which may be valuable if it may not actually get formatted). If *capture_locals* is `True` the local variables in each `FrameSummary` are captured as object representations.

Modifié dans la version 3.12 : Exceptions raised from `repr()` on a local variable (when *capture_locals* is `True`) are no longer propagated to the caller.

classmethod from_list(*a_list*)

Construct a `StackSummary` object from a supplied list of `FrameSummary` objects or old-style list of tuples. Each tuple should be a 4-tuple with *filename*, *lineno*, *name*, *line* as the elements.

format()

Returns a list of strings ready for printing. Each string in the resulting list corresponds to a single frame from the stack. Each string ends in a newline; the strings may contain internal newlines as well, for those items with source text lines.

For long sequences of the same frame and line, the first few repetitions are shown, followed by a summary line stating the exact number of further repetitions.

Modifié dans la version 3.6 : Long sequences of repeated frames are now abbreviated.

format_frame_summary(*frame_summary*)

Returns a string for printing one of the frames involved in the stack. This method is called for each `FrameSummary` object to be printed by `StackSummary.format()`. If it returns `None`, the frame is omitted from the output.

Ajouté dans la version 3.11.

30.11.4 `FrameSummary` Objects

Ajouté dans la version 3.5.

A `FrameSummary` object represents a single frame in a traceback.

class traceback.**FrameSummary**(*filename*, *lineno*, *name*, *lookup_line=True*, *locals=None*, *line=None*)

Represents a single frame in the traceback or stack that is being formatted or printed. It may optionally have a stringified version of the frame's locals included in it. If *lookup_line* is `False`, the source code is not looked up until the `FrameSummary` has the `line` attribute accessed (which also happens when casting it to a `tuple`). `line` may be directly provided, and will prevent line lookups happening at all. *locals* is an optional local variable mapping, and if supplied the variable representations are stored in the summary for later display.

`FrameSummary` instances have the following attributes :

filename

The filename of the source code for this frame. Equivalent to accessing `f.f_code.co_filename` on a frame object *f*.

lineno

The line number of the source code for this frame.

name

Equivalent to accessing `f.f_code.co_name` on a frame object *f*.

line

A string representing the source code for this frame, with leading and trailing whitespace stripped. If the source is not available, it is `None`.

30.11.5 Examples of Using the Module-Level Functions

This simple example implements a basic read-eval-print loop, similar to (but less useful than) the standard Python interactive interpreter loop. For a more complete implementation of the interpreter loop, refer to the `code` module.

```
import sys, traceback

def run_user_code(envdir):
    source = input(">>> ")
    try:
        exec(source, envdir)
    except Exception:
        print("Exception in user code:")
        print("-"*60)
        traceback.print_exc(file=sys.stdout)
        print("-"*60)

envdir = {}
while True:
    run_user_code(envdir)
```

The following example demonstrates the different ways to print and format the exception and traceback :

```
import sys, traceback

def lumberjack():
    bright_side_of_life()

def bright_side_of_life():
    return tuple()[0]

try:
    lumberjack()
except IndexError as exc:
    print("*** print_tb:")
    traceback.print_tb(exc.__traceback__, limit=1, file=sys.stdout)
```

(suite sur la page suivante)

(suite de la page précédente)

```
    print("*** print_exception:")
    traceback.print_exception(exc, limit=2, file=sys.stdout)
    print("*** print_exc:")
    traceback.print_exc(limit=2, file=sys.stdout)
    print("*** format_exc, first and last line:")
    formatted_lines = traceback.format_exc().splitlines()
    print(formatted_lines[0])
    print(formatted_lines[-1])
    print("*** format_exception:")
    print(repr(traceback.format_exception(exc)))
    print("*** extract_tb:")
    print(repr(traceback.extract_tb(exc.__traceback__)))
    print("*** format_tb:")
    print(repr(traceback.format_tb(exc.__traceback__)))
    print("*** tb_lineno:", exc.__traceback__.tb_lineno)
```

The output for the example would look similar to this :

```
*** print_tb:
  File "<doctest...>", line 10, in <module>
    lumberjack()
    ~~~~~~~~~~^^
*** print_exception:
Traceback (most recent call last):
  File "<doctest...>", line 10, in <module>
    lumberjack()
    ~~~~~~~~~~^^
  File "<doctest...>", line 4, in lumberjack
    bright_side_of_life()
    ~~~~~~~~~~~~~~~~~~~^^
IndexError: tuple index out of range
*** print_exc:
Traceback (most recent call last):
  File "<doctest...>", line 10, in <module>
    lumberjack()
    ~~~~~~~~~~^^
  File "<doctest...>", line 4, in lumberjack
    bright_side_of_life()
    ~~~~~~~~~~~~~~~~~~~^^
IndexError: tuple index out of range
*** format_exc, first and last line:
Traceback (most recent call last):
IndexError: tuple index out of range
*** format_exception:
['Traceback (most recent call last):\n',
 '  File "<doctest default[0]>", line 10, in <module>\n    lumberjack()\n    ~~~~~~
↪~~~~^^\n',
 '  File "<doctest default[0]>", line 4, in lumberjack\n    bright_side_of_life()\
↪n    ~~~~~~~~~~~~~~~~~~~^^\n',
 '  File "<doctest default[0]>", line 7, in bright_side_of_life\n    return␣
↪tuple()[0]\n           ~~~~~~~^^^\n',
 'IndexError: tuple index out of range\n']
*** extract_tb:
[<FrameSummary file <doctest...>, line 10 in <module>>,
 <FrameSummary file <doctest...>, line 4 in lumberjack>,
 <FrameSummary file <doctest...>, line 7 in bright_side_of_life>]
```

(suite sur la page suivante)

(suite de la page précédente)

```
*** format_tb:
['  File "<doctest default[0]>", line 10, in <module>\n    lumberjack()\n    ~~~~~~
↪~~~~^^\n',
 '  File "<doctest default[0]>", line 4, in lumberjack\n    bright_side_of_life()\
↪n    ~~~~~~~~~~~~~~~~~~^^\n',
 '  File "<doctest default[0]>", line 7, in bright_side_of_life\n    return␣
↪tuple()[0]\n           ~~~~~~~^^^\n']
*** tb_lineno: 10
```

The following example shows the different ways to print and format the stack :

```
>>> import traceback
>>> def another_function():
...     lumberstack()
...
>>> def lumberstack():
...     traceback.print_stack()
...     print(repr(traceback.extract_stack()))
...     print(repr(traceback.format_stack()))
...
>>> another_function()
  File "<doctest>", line 10, in <module>
    another_function()
  File "<doctest>", line 3, in another_function
    lumberstack()
  File "<doctest>", line 6, in lumberstack
    traceback.print_stack()
[('<doctest>', 10, '<module>', 'another_function()'),
 ('<doctest>', 3, 'another_function', 'lumberstack()'),
 ('<doctest>', 7, 'lumberstack', 'print(repr(traceback.extract_stack()))')]
['  File "<doctest>", line 10, in <module>\n    another_function()\n',
 '  File "<doctest>", line 3, in another_function\n    lumberstack()\n',
 '  File "<doctest>", line 8, in lumberstack\n    print(repr(traceback.format_
↪stack()))\n']
```

This last example demonstrates the final few formatting functions :

```
>>> import traceback
>>> traceback.format_list([('spam.py', 3, '<module>', 'spam.eggs()'),
...                        ('eggs.py', 42, 'eggs', 'return "bacon"')])
['  File "spam.py", line 3, in <module>\n    spam.eggs()\n',
 '  File "eggs.py", line 42, in eggs\n    return "bacon"\n']
>>> an_error = IndexError('tuple index out of range')
>>> traceback.format_exception_only(an_error)
['IndexError: tuple index out of range\n']
```

30.11.6 Examples of Using `TracebackException`

With the helper class, we have more options :

```
>>> import sys
>>> from traceback import TracebackException
>>>
>>> def lumberjack():
...     bright_side_of_life()
...
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> def bright_side_of_life():
...     t = "bright", "side", "of", "life"
...     return t[5]
...
>>> try:
...     lumberjack()
... except IndexError as e:
...     exc = e
...
>>> try:
...     try:
...         lumberjack()
...     except:
...         1/0
... except Exception as e:
...     chained_exc = e
...
>>> # limit works as with the module-level functions
>>> TracebackException.from_exception(exc, limit=-2).print()
Traceback (most recent call last):
  File "<python-input-1>", line 6, in lumberjack
    bright_side_of_life()
    ~~~~~~~~~~~~~~~~~~~^^
  File "<python-input-1>", line 10, in bright_side_of_life
    return t[5]
           ~^^^
IndexError: tuple index out of range

>>> # capture_locals adds local variables in frames
>>> TracebackException.from_exception(exc, limit=-2, capture_locals=True).print()
Traceback (most recent call last):
  File "<python-input-1>", line 6, in lumberjack
    bright_side_of_life()
    ~~~~~~~~~~~~~~~~~~~^^
  File "<python-input-1>", line 10, in bright_side_of_life
    return t[5]
           ~^^^
    t = ("bright", "side", "of", "life")
IndexError: tuple index out of range

>>> # The *chain* kwarg to print() controls whether chained
>>> # exceptions are displayed
>>> TracebackException.from_exception(chained_exc).print()
Traceback (most recent call last):
  File "<python-input-19>", line 4, in <module>
    lumberjack()
    ~~~~~~~~~~^^
  File "<python-input-8>", line 7, in lumberjack
    bright_side_of_life()
    ~~~~~~~~~~~~~~~~~~~^^
  File "<python-input-8>", line 11, in bright_side_of_life
    return t[5]
           ~^^^
IndexError: tuple index out of range

During handling of the above exception, another exception occurred:
```

(suite sur la page suivante)

(suite de la page précédente)

```
Traceback (most recent call last):
  File "<python-input-19>", line 6, in <module>
    1/0
    ~^~
ZeroDivisionError: division by zero

>>> TracebackException.from_exception(chained_exc).print(chain=False)
Traceback (most recent call last):
  File "<python-input-19>", line 6, in <module>
    1/0
    ~^~
ZeroDivisionError: division by zero
```

30.12 __future__ --- Future statement definitions

Source code : Lib/_future_.py

Imports of the form `from __future__ import feature` are called future statements. These are special-cased by the Python compiler to allow the use of new Python features in modules containing the future statement before the release in which the feature becomes standard.

While these future statements are given additional special meaning by the Python compiler, they are still executed like any other import statement and the *__future__* exists and is handled by the import system the same way any other Python module would be. This design serves three purposes :

- éviter de dérouter les outils existants qui analysent les instructions d'importation et s'attendent à trouver les modules qu'ils importent ;
- Pour documenter le phasage de changements entraînant des incompatibilités : introduction, utilisation obligatoire. Il s'agit d'une forme de documentation exécutable, qui peut être inspectée par un programme en important *__future__* et en examinant son contenu.
- To ensure that future statements run under releases prior to Python 2.1 at least yield runtime exceptions (the import of *__future__* will fail, because there was no module of that name prior to 2.1).

30.12.1 Contenu du module

Aucune fonctionnalité ne sera jamais supprimée de *__future__*. Depuis son introduction dans Python 2.1, les fonctionnalités suivantes ont trouvé leur places dans le langage utilisant ce mécanisme :

fonctionnalité	optionnel dans	obligatoire dans	effet
nested_scopes	2.1.0b1	2.2	**PEP 227** : *Portées imbriquées*
générateurs	2.2.0a1	2.3	**PEP 255** : *Générateurs simples*
division	2.2.0a2	3.0	**PEP 238** : *Changement de l'opérateur de division*
absolute_import	2.5.0a1	3.0	**PEP 328** : *Importations : multilignes et absolues/relatives* (ressource en anglais)
with_statem	2.5.0a1	2.6	**PEP 343** : *L'instruction "with"*
print_functi	2.6.0a2	3.0	**PEP 3105** : *Transformation de print en fonction*
unicode_literal	2.6.0a2	3.0	**PEP 3112** : *Chaînes d'octets littéraux en Python 3000*
generator_stop	3.5.0b1	3.7	**PEP 479** : *Gestion de *StopIteration* à l'intérieur des générateurs**
annotations	3.7.0b1	Never [1]	**PEP 563** : *Postponed evaluation of annotations*, **PEP 649** : *Deferred evaluation of annotations using descriptors*

class `__future__.`**`_Feature`**

Chaque instruction dans `__future__.py` est de la forme :

```
FeatureName = _Feature(OptionalRelease, MandatoryRelease,
                       CompilerFlag)
```

où, normalement, *OptionalRelease* est inférieur à *MandatoryRelease*, et les deux sont des quintuplets de la même forme que `sys.version_info` :

```
(PY_MAJOR_VERSION, # the 2 in 2.1.0a3; an int
 PY_MINOR_VERSION, # the 1; an int
 PY_MICRO_VERSION, # the 0; an int
 PY_RELEASE_LEVEL, # "alpha", "beta", "candidate" or "final"; string
 PY_RELEASE_SERIAL # the 3; an int
)
```

`_Feature.`**`getOptionalRelease`**`()`

OptionalRelease enregistre la première version dans laquelle la fonctionnalité a été acceptée.

`_Feature.`**`getMandatoryRelease`**`()`

Dans le cas d'un *MandatoryRelease* qui n'a pas encore eu lieu, *MandatoryRelease* prédit la *release* dans laquelle la fonctionnalité deviendra un élément du langage.

Sinon *MandatoryRelease* enregistre lorsque la fonctionnalité est devenue une partie du langage ; dans cette version ou les suivantes, les modules n'ont plus besoin d'une déclaration *future* pour utiliser la fonctionnalité en question, mais ils peuvent continuer à utiliser ces importations.

MandatoryRelease may also be `None`, meaning that a planned feature got dropped or that it is not yet decided.

`_Feature.`**`compiler_flag`**

CompilerFlag is the (bitfield) flag that should be passed in the fourth argument to the built-in function `compile()` to enable the feature in dynamically compiled code. This flag is stored in the `_Feature.compiler_flag` attribute on `_Feature` instances.

Voir aussi

future
Comment le compilateur gère les importations « futures ».

PEP 236 — retour vers le `__future__`
La proposition originale pour le mécanisme de `__future__`.

30.13 `gc` --- Garbage Collector interface

Ce module constitue une interface au ramasse-miettes facultatif. Il permet de désactiver le ramasse-miettes ou de régler la fréquence des passages. Il fournit des options de débogage, et donne aussi accès aux objets qui ne peuvent pas être détruits bien qu'ils aient été détectés comme non référencés. Le ramasse-miettes vient en complément du système de comptage de références, et peut donc être désactivé pour du code qui ne crée aucun cycle de références. On le désactive avec `gc.disable()`. Pour remonter à la source d'une fuite de mémoire, utilisez `gc.set_debug(gc.DEBUG_LEAK)`. Notez que `gc.DEBUG_LEAK` inclut `gc.DEBUG_SAVEALL`. Cette dernière option fait que les objets inatteignables, au lieu d'être détruits, sont placés dans la liste `gc.garbage` pour pouvoir y être examinés.

Le module `gc` contient les fonctions suivantes :

1. `from __future__ import annotations` was previously scheduled to become mandatory in Python 3.10, but the change was delayed and ultimately canceled. This feature will eventually be deprecated and removed. See PEP 649 and PEP 749.

`gc.enable()`

Active le ramasse-miettes.

`gc.disable()`

Désactive le ramasse-miettes.

`gc.isenabled()`

Renvoie `True` ou `False` selon que le ramasse-miettes est activé ou non.

`gc.collect(generation=2)`

Perform a collection. The optional argument *generation* may be an integer specifying which generation to collect (from 0 to 2). A `ValueError` is raised if the generation number is invalid. The sum of collected objects and uncollectable objects is returned.

Calling `gc.collect(0)` will perform a GC collection on the young generation.

Calling `gc.collect(1)` will perform a GC collection on the young generation and an increment of the old generation.

Calling `gc.collect(2)` or `gc.collect()` performs a full collection

The free lists maintained for a number of built-in types are cleared whenever a full collection or collection of the highest generation (2) is run. Not all items in some free lists may be freed due to the particular implementation, in particular `float`.

The effect of calling `gc.collect()` while the interpreter is already performing a collection is undefined.

Modifié dans la version 3.13 : `generation=1` performs an increment of collection.

`gc.set_debug(flags)`

Change les options de débogage du ramasse-miettes, qui activent l'écriture d'informations sur `sys.stderr`. Une liste d'options se trouve plus bas. Les options peuvent se combiner par les opérateurs bit à bit.

`gc.get_debug()`

Renvoie les options de débogage actives.

`gc.get_objects(generation=None)`

Returns a list of all objects tracked by the collector, excluding the list returned. If *generation* is not `None`, return only the objects as follows :
— 0 : All objects in the young generation
— 1 : No objects, as there is no generation 1 (as of Python 3.13)
— 2 : All objects in the old generation
Modifié dans la version 3.8 : ajout du paramètre *generation*.

Modifié dans la version 3.13 : Generation 1 is removed

Lève un *événement d'audit* `gc.get_objects` avec l'argument `generation`.

`gc.get_stats()`

Renvoie une liste de trois dictionnaires, un par génération. Ils contiennent des statistiques sur l'action du ramasse-miettes depuis le lancement de l'interpréteur. Les clés actuellement présentes sont les suivantes (d'autres pourraient être ajoutées dans des versions ultérieures) :
— `collections`, le nombre de fois où cette génération a été examinée par le ramasse-miettes ;
— `collected`, le nombre total d'objets qui ont été détruits alors qu'ils étaient dans cette génération ;
— `uncollectable`, le nombre total d'objets qui ont été identifiés comme indestructibles (et donc ajoutés à la liste `garbage`) au sein de cette génération.
Ajouté dans la version 3.4.

`gc.set_threshold(threshold0[, threshold1[, threshold2]])`

Règle les seuils de déclenchement du ramasse-miettes, qui déterminent sa fréquence de passage. Si *threshold0* est mis à zéro, le ramasse-miettes ne passe jamais.

The GC classifies objects into two generations depending on whether they have survived a collection. New objects are placed in the young generation. If an object survives a collection it is moved into the old generation.

In order to decide when to run, the collector keeps track of the number of object allocations and deallocations since the last collection. When the number of allocations minus the number of deallocations exceeds *threshold0*, collection starts. For each collection, all the objects in the young generation and some fraction of the old generation is collected.

The fraction of the old generation that is collected is **inversely** proportional to *threshold1*. The larger *threshold1* is, the slower objects in the old generation are collected. For the default value of 10, 1% of the old generation is scanned during each collection.

threshold2 is ignored.

See Garbage collector design for more information.

Modifié dans la version 3.13 : *threshold2* is ignored

`gc.get_count()`

Renvoie un triplet des nombres totaux de passages effectués par génération.

`gc.get_threshold()`

Renvoie les seuils de passage sous la forme du triplet `(threshold0, threshold1, threshold2)`.

`gc.get_referrers(*objs)`

Renvoie la liste des objets qui contiennent directement une référence à l'un quelconque des arguments. Il est à noter que cette fonction prend uniquement en compte les objets suivis par le ramasse-miettes, ce qui exclut les instances de certains types d'extension qui contiennent bien des références sans pour autant prendre en charge le ramassage des miettes.

La liste renvoyée peut contenir des objets déjà isolés, mais maintenus en mémoire à cause d'un cycle. Pour les exclure, appelez `collect()` juste avant `get_referrers()`.

> **Avertissement**
>
> La manipulation des objets renvoyés par `get_referrers()` est hasardeuse car ils risquent d'être encore en cours d'initialisation, donc dans un état temporairement instable. Mieux vaut réserver `get_referrers()` au débogage.

Lève un *événement d'audit* `gc.get_referrers` avec l'argument `objs`.

`gc.get_referents(*objs)`

Renvoie une liste des objets pointés par les références que contiennent les arguments. Ils sont déterminés en appelant, si présente, la méthode C `tp_traverse` de chaque argument, qui visite les objets auxquels cet argument fait référence. Il est à noter que `tp_traverse` n'est définie que par les objets qui gèrent le ramassage des miettes, et n'a l'obligation de visiter que les objets qui peuvent potentiellement faire partie d'un cycle. Ainsi, la liste renvoyée par cette fonction ne contient par forcément tous les objets qu'il est possible d'atteindre à partir des arguments. Par exemple, si l'un des arguments contient un entier, ce dernier objet peut être présent ou non dans la liste.

Lève un *événement d'audit* `gc.get_referents` avec l'argument `objs`.

`gc.is_tracked(obj)`

Renvoie `True` ou `False` selon que l'argument est suivi ou non par le ramasse-miettes. En règle générale, les objets atomiques ne sont pas suivis, tandis que les objets non-atomiques, tels que les conteneurs et instances de classes définies par l'utilisateur, le sont. Cependant, certains types présentent des optimisations qui permettent de se passer avantageusement du ramasse-miettes dans les cas simples, comme les dictionnaires dont toutes les clés et valeurs sont atomiques :

```
>>> gc.is_tracked(0)
False
>>> gc.is_tracked("a")
False
>>> gc.is_tracked([])
```

(suite sur la page suivante)

(suite de la page précédente)

```
True
>>> gc.is_tracked({})
False
>>> gc.is_tracked({"a": 1})
True
```

Ajouté dans la version 3.1.

`gc.is_finalized(obj)`

Renvoie `True` ou `False` selon que l'argument a été finalisé par le ramasse-miettes.

```
>>> x = None
>>> class Lazarus:
...     def __del__(self):
...         global x
...         x = self
...
>>> lazarus = Lazarus()
>>> gc.is_finalized(lazarus)
False
>>> del lazarus
>>> gc.is_finalized(x)
True
```

Ajouté dans la version 3.9.

`gc.freeze()`

Freeze all the objects tracked by the garbage collector ; move them to a permanent generation and ignore them in all the future collections.

If a process will `fork()` without `exec()`, avoiding unnecessary copy-on-write in child processes will maximize memory sharing and reduce overall memory usage. This requires both avoiding creation of freed "holes" in memory pages in the parent process and ensuring that GC collections in child processes won't touch the `gc_refs` counter of long-lived objects originating in the parent process. To accomplish both, call `gc.disable()` early in the parent process, `gc.freeze()` right before `fork()`, and `gc.enable()` early in child processes.

Ajouté dans la version 3.7.

`gc.unfreeze()`

Unfreeze the objects in the permanent generation, put them back into the oldest generation.

Ajouté dans la version 3.7.

`gc.get_freeze_count()`

Return the number of objects in the permanent generation.

Ajouté dans la version 3.7.

Les variables suivantes sont publiques, mais elles ne sont pas censées être modifiées (vous pouvez les muter, mais pas les redéfinir) :

`gc.garbage`

Liste des objets indestructibles, que le ramasse-miettes n'a pas pu éliminer bien qu'ils soient inatteignables. Depuis Python 3.4, cette liste demeure la plupart du temps vide. Elle peut se remplir si le programme fait appel à des types d'extension définis en C avec un champ `tp_del` différent de `NULL`.

Si `DEBUG_SAVEALL` est actif, tous les objets inatteignables sont ajoutés à cette liste au lieu d'être détruits.

Modifié dans la version 3.2 : Si cette liste n'est pas vide lors de l'*arrêt de l'interpréteur*, un `ResourceWarning` est émis (les avertissements de ce type sont silencieux par défaut). De plus, si `DEBUG_UNCOLLECTABLE` est actif, tous les objets indestructibles sont affichés.

Modifié dans la version 3.4 : Following **PEP 442**, objects with a `__del__()` method don't end up in `gc.garbage` anymore.

`gc.callbacks`

Liste de fonctions de rappel lancées par le ramasse-miettes avant et après un passage. Elles prennent deux arguments, *phase* et *info*.

phase peut prendre deux valeurs :

`"start"` lorsque le passage du ramasse-miettes est imminent.

`"stop"` lorsque le passage du ramasse-miettes vient de se terminer.

info est un dictionnaire qui donne plus d'informations à la fonction de rappel. Les clés suivantes sont actuellement présentes :

`"generation"`, la génération la plus âgée intégrée à ce passage ;

`"collected"` : si *phase* vaut `"stop"`, le nombre d'objets détruits avec succès ;

`"uncollectable"` : si *phase* vaut `"stop"`, le nombre d'objets indestructibles ajoutés à `garbage`.

Toute application peut ajouter ses propres fonctions de rappel à cette liste. Voici les principales applications :

Faire des statistiques sur le passage du ramasse-miettes, par exemple la fréquence à laquelle chaque génération est examinée, ou bien le temps d'un passage ;

Identifier les types définis par une application dont les instances s'ajoutent à `garbage` car elles sont indestructibles.

Ajouté dans la version 3.3.

Les constantes suivantes définissent les options de débogage que l'on peut passer à `set_debug()` :

`gc.DEBUG_STATS`

Affiche des statistiques durant les passages du ramasse-miettes. Utile pour pouvoir régler la fréquence des passages.

`gc.DEBUG_COLLECTABLE`

Affiche des informations sur les objets détruits.

`gc.DEBUG_UNCOLLECTABLE`

Affiche des informations sur les objets indestructibles (ceux qui sont ajoutés à la liste `garbage`, qui sont inatteignables mais dont la mémoire ne peut pas être libérée).

Modifié dans la version 3.2 : Affiche également le contenu de `garbage` à l'*arrêt de l'interpréteur*, pour peu que cette liste ne soit pas vide.

`gc.DEBUG_SAVEALL`

Lorsque cette option est active, les objets inatteignables sont ajoutés à la liste *garbage* au lieu d'être supprimés. Ceci est utile pour déboguer une fuite de mémoire.

`gc.DEBUG_LEAK`

Combinaison des options utiles au débogage d'une fuite de mémoire. Il s'agit d'un raccourci pour `DEBUG_COLLECTABLE|DEBUG_UNCOLLECTABLE|DEBUG_SAVEALL`.

30.14 `inspect` --- Inspect live objects

Code source : Lib/inspect.py

Le module `inspect` implémente plusieurs fonctions permettant d'obtenir de l'information à propos d'objets tels que les modules, classes, méthodes, fonctions, traces de piles, cadres d'exécution et objets code. Par exemple, il permet d'inspecter le contenu d'une classe, de retrouver le code source d'une méthode, d'extraire et de mettre en forme une liste d'arguments pour une fonction, ou obtenir toute l'information nécessaire pour afficher une pile d'appels détaillée.

Ce module donne accès à 4 types de services : vérification de typage, obtention de code source, inspection de classes et de fonctions et examen de la pile d'exécution.

30.14.1 Types et membres

The `getmembers()` function retrieves the members of an object such as a class or module. The functions whose names begin with "is" are mainly provided as convenient choices for the second argument to `getmembers()`. They also help you determine when you can expect to find the following special attributes (see import-mod-attrs for module attributes) :

Type	Attribut	Description
classe	__doc__	texte de documentation
	__name__	nom de la classe lors de sa définition
	__qualname__	nom qualifié
	__module__	nom du module contenant la classe lors de sa définition
	__type_params__	A tuple containing the type parameters of a generic class
méthode	__doc__	texte de documentation
	__name__	nom de la méthode lors de sa définition
	__qualname__	nom qualifié
	__func__	objet fonction contenant l'implémentation de la méthode
	__self__	instance à laquelle cette méthode est liée, ou `None`
	__module__	nom du module contenant la méthode lors de sa définition
fonction	__doc__	texte de documentation
	__name__	nom de la fonction lors de sa définition
	__qualname__	nom qualifié
	__code__	objet code contenant le *code intermédiaire* d'une fonction compilée
	__defaults__	*n*-uplet de valeurs par défaut pour arguments positionnels ou nommés
	__kwdefaults__	tableau de correspondance de valeurs par défaut pour les paramètres nommés
	__globals__	espace de nommage global lors de la définition de la fonction
	__builtins__	builtins namespace
	__annotations__	tableau de correspondance des paramètres à leur annotation ; `"return"` est un mot-cl
	__type_params__	A tuple containing the type parameters of a generic function
	__module__	nom du module contenant la fonction lors de sa définition
traceback	tb_frame	objet cadre à ce niveau
	tb_lasti	index de la dernière instruction lancée dans le code intermédiaire (`bytecode`)
	tb_lineno	numéro de ligne courant dans le code source Python
	tb_next	trace de pile interne suivante (appel par le cadre courant)
cadre	f_back	trace d'exécution externe suivante (le contexte appelant du cadre courant)
	f_builtins	espace de nommage natif du cadre courant
	f_code	objet code exécuté dans le cadre courant
	f_globals	espace de nommage global tel que vu par cadre courant
	f_lasti	index de la dernière instruction lancée dans le code intermédiaire (`bytecode`)
	f_lineno	numéro de ligne courant dans le code source Python
	f_locals	espace de nommage local du cadre courant
	f_generator	returns the generator or coroutine object that owns this frame, or `None` if the frame is
	f_trace	fonction de traçage du cadre courant, ou `None`
	f_trace_lines	indicate whether a tracing event is triggered for each source source line
	f_trace_opcodes	indicate whether per-opcode events are requested
	clear()	used to clear all references to local variables
code	co_argcount	nombre d'arguments (excluant les arguments *keyword-only*, * ou **)
	co_code	texte du code intermédiaire compilé
	co_cellvars	*n*-uplet du nom des cellules variables (référencées par leur contexte)
	co_consts	*n*-uplet des constantes utilisées dans le code intermédiaire (`bytecode`)
	co_filename	nom du fichier dans lequel cet objet code a été créé
	co_firstlineno	numéro de la première ligne dans le code source Python
	co_flags	**bitmap** des options `CO_*`, voir plus *ici*
	co_lnotab	tableau de correspondance encodée des numéros de ligne aux numéros d'indices du co
	co_freevars	*n*-uplet des noms de variables libres (référencées par la fermeture (*closure* en anglais)
	co_posonlyargcount	nombre d'arguments positionnels
	co_kwonlyargcount	nombre d'arguments *keywords-only* (excluant les arguments **)

Tableau 2 – suite de la page précédente

Type	Attribut	Description
	co_name	nom de l'objet code lors de sa définition
	co_qualname	fully qualified name with which this code object was defined
	co_names	tuple of names other than arguments and function locals
	co_nlocals	nombre de variables locales
	co_stacksize	espace requis pour la pile de la machine virtuelle
	co_varnames	*n*-uplet des noms des arguments et variables locales
	co_lines()	returns an iterator that yields successive bytecode ranges
	co_positions()	returns an iterator of source code positions for each bytecode instruction
	replace()	returns a copy of the code object with new values
generator	__name__	nom
	__qualname__	nom qualifié
	gi_frame	cadre
	gi_running	est-ce que le générateur est en cours d'exécution ?
	gi_code	code
	gi_yieldfrom	objet pointé par l'itérateur `yield from`, ou `None`
`async generator`	__name__	nom
	__qualname__	nom qualifié
	`cr_await`	objet sur lequel on attend (*await*) présentement, ou `None`
	`gi_frame`	cadre
	`ag_running`	est-ce que le générateur est en cours d'exécution ?
	`ag_code`	code
coroutine	__name__	nom
	__qualname__	nom qualifié
	`cr_await`	objet sur lequel on attend (*await*) présentement, ou `None`
	`cr_frame`	cadre
	`cr_running`	est-ce que la coroutine est en cours d'exécution ?
	`cr_code`	code
	`cr_origin`	lieu de création de la coroutine, ou `None`. Voir `sys.set_coroutine_origin_tra`
builtin	__doc__	texte de documentation
	__name__	nom originel de cette fonction ou méthode
	__qualname__	nom qualifié
	__self__	instance à laquelle est liée une méthode, ou `None`

Modifié dans la version 3.5 : ajout des attributs `__qualname` et `gi_yieldfrom` aux générateurs.

L'attribut `__name__` du générateur est maintenant déterminé par le nom de la fonction, plutôt que le nom du code, et peut maintenant être modifié.

Modifié dans la version 3.7 : Ajout de l'attribut `cr_origin` aux coroutines.

Modifié dans la version 3.10 : Add `__builtins__` attribute to functions.

Modifié dans la version 3.14 : Add `f_generator` attribute to frames.

`inspect.`**`getmembers`**(*object*[, *predicate*])

Renvoie tous les membres d'un objet sous la forme d'une liste de paires `(nom, valeur)`, ordonnée par nom. Si l'argument *predicate* (qui sera appelé avec la `valeur` de chaque membre) est fourni, seuls les membres pour lequel le prédicat est vrai seront inclus.

> **Note**
>
> `getmembers()` will only return class attributes defined in the metaclass when the argument is a class and those attributes have been listed in the metaclass' custom `__dir__()`.

`inspect.`**`getmembers_static`**(*object*[, *predicate*])

Return all the members of an object in a list of `(name, value)` pairs sorted by name without triggering

dynamic lookup via the descriptor protocol, __getattr__ or __getattribute__. Optionally, only return members that satisfy a given predicate.

> **Note**
>
> `getmembers_static()` may not be able to retrieve all members that getmembers can fetch (like dynamically created attributes) and may find members that getmembers can't (like descriptors that raise AttributeError). It can also return descriptor objects instead of instance members in some cases.

Ajouté dans la version 3.11.

`inspect.getmodulename(path)`

Renvoie le nom du module indiqué par le fichier *path*, excluant le nom des paquets englobants. L'extension du fichier est vérifiée parmi toutes les entrées renvoyées par `importlib.machinery.all_suffixes()`. S'il y a une correspondance, le chemin final est renvoyé, sans l'extension. Autrement, `None` est renvoyé.

Notez que cette fonction renvoie un nom significatif *seulement* pour les modules Python - les chemins qui réfèrent à des paquets Python donnent toujours `None`.

Modifié dans la version 3.3 : La fonction est basée directement sur `importlib`.

`inspect.ismodule(object)`

Renvoie `True` si l'objet est un module.

`inspect.isclass(object)`

Renvoie `True` si l'objet est une classe, qu'il s'agisse d'un objet natif ou qu'elle soit écrite en Python.

`inspect.ismethod(object)`

Renvoie `True` si l'objet est une méthode liée écrite en Python.

`inspect.ispackage(object)`

Return `True` if the object is a *package*.

Ajouté dans la version 3.14.

`inspect.isfunction(object)`

Renvoie `True` si l'objet est une fonction Python, ce qui inclut les fonctions créées par une expression *lambda*.

`inspect.isgeneratorfunction(object)`

Renvoie `True` si l'objet est un générateur.

Modifié dans la version 3.8 : les fonctions englobées dans `functools.partial()` renvoient maintenant `True` si la fonction englobée est un générateur Python.

Modifié dans la version 3.13 : Functions wrapped in `functools.partialmethod()` now return `True` if the wrapped function is a Python generator function.

`inspect.isgenerator(object)`

Renvoie `True` si l'objet est un générateur.

`inspect.iscoroutinefunction(object)`

Return `True` if the object is a *coroutine function* (a function defined with an `async def` syntax), a `functools.partial()` wrapping a *coroutine function*, or a sync function marked with `markcoroutinefunction()`.

Ajouté dans la version 3.5.

Modifié dans la version 3.8 : les fonctions englobées par `functools.partial()` renvoient maintenant `True` si la fonction englobée est une *coroutine*.

Modifié dans la version 3.12 : Sync functions marked with `markcoroutinefunction()` now return `True`.

Modifié dans la version 3.13 : Functions wrapped in `functools.partialmethod()` now return `True` if the wrapped function is a *coroutine function*.

inspect.**markcoroutinefunction**(*func*)

Decorator to mark a callable as a *coroutine function* if it would not otherwise be detected by `iscoroutinefunction()`.

This may be of use for sync functions that return a *coroutine*, if the function is passed to an API that requires `iscoroutinefunction()`.

When possible, using an `async def` function is preferred. Also acceptable is calling the function and testing the return with `iscoroutine()`.

Ajouté dans la version 3.12.

inspect.**iscoroutine**(*object*)

Renvoie `True` si l'objet est une *coroutine* créée par une fonction `async def`.

Ajouté dans la version 3.5.

inspect.**isawaitable**(*object*)

Renvoie `True` si l'objet peut être utilisé par une expression `await`.

Can also be used to distinguish generator-based coroutines from regular generators :

```
import types

def gen():
    yield
@types.coroutine
def gen_coro():
    yield

assert not isawaitable(gen())
assert isawaitable(gen_coro())
```

Ajouté dans la version 3.5.

inspect.**isasyncgenfunction**(*object*)

Return `True` if the object is an *asynchronous generator* function, for example :

```
>>> async def agen():
...     yield 1
...
>>> inspect.isasyncgenfunction(agen)
True
```

Ajouté dans la version 3.6.

Modifié dans la version 3.8 : Functions wrapped in `functools.partial()` now return `True` if the wrapped function is an *asynchronous generator* function.

Modifié dans la version 3.13 : Functions wrapped in `functools.partialmethod()` now return `True` if the wrapped function is a *coroutine function*.

inspect.**isasyncgen**(*object*)

Renvoie `True` si l'objet est un *itérateur de générateur asynchrone* créé par une fonction *génératrice asynchrone*.

Ajouté dans la version 3.6.

inspect.**istraceback**(*object*)

Renvoie `True` si l'objet est une trace d'exécution.

inspect.**isframe**(*object*)

Renvoie `True` si l'objet est un cadre.

`inspect.iscode(object)`

Renvoie `True` si l'objet est un code.

`inspect.isbuiltin(object)`

Renvoie `True` si l'objet est une fonction native ou une méthode liée native.

`inspect.ismethodwrapper(object)`

Return `True` if the type of object is a *`MethodWrapperType`*.

These are instances of *`MethodWrapperType`*, such as `__str__()`, `__eq__()` and `__repr__()`.

Ajouté dans la version 3.11.

`inspect.isroutine(object)`

Renvoie `True` si l'objet est une fonction, qu'elle soit définie par l'utilisateur, native, ou une méthode.

`inspect.isabstract(object)`

Renvoie `True` si l'objet est une classe de base abstraite.

`inspect.ismethoddescriptor(object)`

Renvoie `True` si l'objet est un descripteur de méthode sauf si *`ismethod()`*, *`isclass()`*, *`isfunction()`* ou *`isbuiltin()`* sont vrais (`True`).

This, for example, is true of `int.__add__`. An object passing this test has a `__get__()` method, but not a `__set__()` method or a `__delete__()` method. Beyond that, the set of attributes varies. A *`__name__`* attribute is usually sensible, and *`__doc__`* often is.

Methods implemented via descriptors that also pass one of the other tests return `False` from the *`ismethoddescriptor()`* test, simply because the other tests promise more -- you can, e.g., count on having the `__func__` attribute (etc) when an object passes *`ismethod()`*.

Modifié dans la version 3.13 : This function no longer incorrectly reports objects with `__get__()` and `__delete__()`, but not `__set__()`, as being method descriptors (such objects are data descriptors, not method descriptors).

`inspect.isdatadescriptor(object)`

Renvoie `True` si l'objet est un descripteur de données.

Data descriptors have a `__set__` or a `__delete__` method. Examples are properties (defined in Python), getsets, and members. The latter two are defined in C and there are more specific tests available for those types, which is robust across Python implementations. Typically, data descriptors will also have *`__name__`* and `__doc__` attributes (properties, getsets, and members have both of these attributes), but this is not guaranteed.

`inspect.isgetsetdescriptor(object)`

Renvoie `True` si l'objet est un descripteur *getset*.

getsets sont des attributs définis dans des extensions de modules via des structures `PyGetSetDef`. Pour les implémentations Python ne possédant pas ces types, cette méthode renvoie toujours `False`.

`inspect.ismemberdescriptor(object)`

Renvoie `True` si l'objet est un membre descripteur.

Les membres descripteurs sont des attributs définis dans des extensions de modules via des structures `PyMemberDef`. Pour les implémentations Python ne possédant pas ces types, cette méthode renvoie toujours `False`.

30.14.2 Récupération du code source

`inspect.getdoc(object)`

Get the documentation string for an object, cleaned up with *`cleandoc()`*. If the documentation string for an object is not provided and the object is a class, a method, a property or a descriptor, retrieve the documentation string from the inheritance hierarchy. Return `None` if the documentation string is invalid or missing.

Modifié dans la version 3.5 : les chaînes de documentation sont maintenant héritées si elles ne sont pas surchargées.

inspect.**getcomments**(*object*)

Renvoie en une chaîne de caractères toute ligne de commentaire qui précède immédiatement le code source de l'objet (pour une classe, fonction ou méthode) ou en début d'un fichier de code source Python (si l'objet est un module). Si le code source de l'objet n'est pas disponible, renvoie `None`. Cela peut se produire si l'objet a été défini en C ou dans un interpréteur interactif.

inspect.**getfile**(*object*)

Renvoie le nom du fichier (texte ou binaire) dans lequel un objet a été défini. Cette méthode échoue avec `TypeError` si l'objet est un module, une classe ou une fonction native.

inspect.**getmodule**(*object*)

Try to guess which module an object was defined in. Return `None` if the module cannot be determined.

inspect.**getsourcefile**(*object*)

Return the name of the Python source file in which an object was defined or `None` if no way can be identified to get the source. This will fail with a `TypeError` if the object is a built-in module, class, or function.

inspect.**getsourcelines**(*object*)

Return a list of source lines and starting line number for an object. The argument may be a module, class, method, function, traceback, frame, or code object. The source code is returned as a list of the lines corresponding to the object and the line number indicates where in the original source file the first line of code was found. An `OSError` is raised if the source code cannot be retrieved. A `TypeError` is raised if the object is a built-in module, class, or function.

Modifié dans la version 3.3 : `OSError` est levé plutôt que `IOError`, qui est maintenant un alias de ce premier.

inspect.**getsource**(*object*)

Return the text of the source code for an object. The argument may be a module, class, method, function, traceback, frame, or code object. The source code is returned as a single string. An `OSError` is raised if the source code cannot be retrieved. A `TypeError` is raised if the object is a built-in module, class, or function.

Modifié dans la version 3.3 : `OSError` est levé plutôt que `IOError`, qui est maintenant un alias de ce premier.

inspect.**cleandoc**(*doc*)

Nettoie les indentations des chaînes de documentation qui sont indentées de manières à s'aligner avec les blocs de code.

Enlève tous les espaces blancs en début de ligne à partir de la première. Tout espace blanc en début de ligne qui puisse être enlevé de manière uniforme à partir de la seconde ligne le sera. Les lignes vides en début et en fin de document seront ensuite enlevées. Les tabulations seront converties en espaces.

30.14.3 Introspection des appelables avec l'objet Signature

Ajouté dans la version 3.3.

The `Signature` object represents the call signature of a callable object and its return annotation. To retrieve a `Signature` object, use the `signature()` function.

inspect.**signature**(*callable*, *, *follow_wrapped=True*, *globals=None*, *locals=None*, *eval_str=False*, *annotation_format=Format.VALUE*)

Return a `Signature` object for the given *callable* :

```
>>> from inspect import signature
>>> def foo(a, *, b:int, **kwargs):
...     pass

>>> sig = signature(foo)

>>> str(sig)
'(a, *, b: int, **kwargs)'
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> str(sig.parameters['b'])
'b: int'

>>> sig.parameters['b'].annotation
<class 'int'>
```

La fonction accepte une grande étendue d'appelables Python, des fonctions et classes jusqu'aux objets `functools.partials()`.

If some of the annotations are strings (e.g., because `from __future__ import annotations` was used), `signature()` will attempt to automatically un-stringize the annotations using `annotationlib.get_annotations()`. The *globals*, *locals*, and *eval_str* parameters are passed into `annotationlib.get_annotations()` when resolving the annotations; see the documentation for `annotationlib.get_annotations()` for instructions on how to use these parameters. A member of the `annotationlib.Format` enum can be passed to the *annotation_format* parameter to control the format of the returned annotations. For example, use `annotation_format=annotationlib.Format.STRING` to return annotations in string format.

Raises `ValueError` if no signature can be provided, and `TypeError` if that type of object is not supported. Also, if the annotations are stringized, and *eval_str* is not false, the `eval()` call(s) to un-stringize the annotations in `annotationlib.get_annotations()` could potentially raise any kind of exception.

A slash (/) in the signature of a function denotes that the parameters prior to it are positional-only. For more info, see the FAQ entry on positional-only parameters.

Modifié dans la version 3.5 : The *follow_wrapped* parameter was added. Pass `False` to get a signature of *callable* specifically (`callable.__wrapped__` will not be used to unwrap decorated callables.)

Modifié dans la version 3.10 : The *globals*, *locals*, and *eval_str* parameters were added.

Modifié dans la version 3.14 : The *annotation_format* parameter was added.

> **Note**
>
> Certains appelables ne peuvent bénéficier d'introspection dans certains environnements Python. Par exemple, en CPython, certaines fonctions natives définies en C ne fournissent aucune métadonnée sur leurs arguments.

Particularité de l'implémentation CPython : If the passed object has a `__signature__` attribute, we may use it to create the signature. The exact semantics are an implementation detail and are subject to unannounced changes. Consult the source code for current semantics.

class `inspect.`**`Signature`**(*parameters=None*, *, *return_annotation=Signature.empty*)

A `Signature` object represents the call signature of a function and its return annotation. For each parameter accepted by the function it stores a `Parameter` object in its `parameters` collection.

L'argument optionnel *parameters* est une séquence d'objets `Parameters`, qui permet de vérifier qu'il n'y a pas de duplication de noms de paramètres, que les paramètres sont dans le bon ordre (positionnels d'abord, positionnels ou nommés ensuite et que les paramètres avec valeurs par défaut sont placés après ceux qui n'en ont pas).

The optional *return_annotation* argument can be an arbitrary Python object. It represents the "return" annotation of the callable.

`Signature` objects are *immutable*. Use `Signature.replace()` or `copy.replace()` to make a modified copy.

Modifié dans la version 3.5 : `Signature` objects are now picklable and *hashable*.

`empty`

Un marqueur spécial, accessible via `Signature.empty`, indiquant l'absence d'une annotation de type

renvoyé.

parameters

Une table de correspondance ordonnée des noms de paramètres avec les objets `Parameter`. Les paramètres apparaissent dans l'ordre strict de leur définition, incluant les paramètres exclusivement nommés.

Modifié dans la version 3.7 : Python ne garantit explicitement la préservation de l'ordre des paramètres exclusivement nommés que depuis la version 3.7, bien qu'en pratique cet ordre a toujours été préservé sous Python 3.

return_annotation

L'annotation du type renvoyé par l'appelable. Si l'appelable n'a pas d'annotation de type, cet attribut est évalué à `Signature.empty`.

bind(**args*, ***kwargs*)

Crée une table de correspondance entre les arguments positionnels et nommés avec les paramètres. Renvoie `BoundArguments` si `*args` et `**kwargs` concordent avec la signature, autrement lève une `TypeError`.

bind_partial(**args*, ***kwargs*)

Fonctionne de manière similaire à `Signature.bind()`, mais permet l'omission de certains arguments (semblable au comportement de `functools.partial()`). Renvoie `BoundArguments`, ou lève une `TypeError` si l'argument passé ne correspond pas à la signature.

replace(**[, parameters][, return_annotation]*)

Create a new `Signature` instance based on the instance `replace()` was invoked on. It is possible to pass different *parameters* and/or *return_annotation* to override the corresponding properties of the base signature. To remove `return_annotation` from the copied `Signature`, pass in `Signature.empty`.

```
>>> def test(a, b):
...     pass
...
>>> sig = signature(test)
>>> new_sig = sig.replace(return_annotation="new return anno")
>>> str(new_sig)
"(a, b) -> 'new return anno'"
```

`Signature` objects are also supported by the generic function `copy.replace()`.

format(**, max_width=None, quote_annotation_strings=True*)

Create a string representation of the `Signature` object.

If *max_width* is passed, the method will attempt to fit the signature into lines of at most *max_width* characters. If the signature is longer than *max_width*, all parameters will be on separate lines.

If *quote_annotation_strings* is False, *annotations* in the signature are displayed without opening and closing quotation marks if they are strings. This is useful if the signature was created with the `STRING` format or if `from __future__ import annotations` was used.

Ajouté dans la version 3.13.

Modifié dans la version 3.14 : The *unquote_annotations* parameter was added.

classmethod from_callable(*obj, *, follow_wrapped=True, globals=None, locals=None, eval_str=False*)

Return a `Signature` (or its subclass) object for a given callable *obj*.

This method simplifies subclassing of `Signature` :

```
class MySignature(Signature):
    pass
sig = MySignature.from_callable(sum)
assert isinstance(sig, MySignature)
```

Its behavior is otherwise identical to that of `signature()`.

Ajouté dans la version 3.5.

Modifié dans la version 3.10 : The *globals*, *locals*, and *eval_str* parameters were added.

class `inspect.`**`Parameter`**(*name*, *kind*, ***, *default=Parameter.empty*, *annotation=Parameter.empty*)

`Parameter` objects are *immutable*. Instead of modifying a `Parameter` object, you can use `Parameter.replace()` or `copy.replace()` to create a modified copy.

Modifié dans la version 3.5 : Parameter objects are now picklable and *hashable*.

empty

Un marqueur spécial, accessible via `Parameters.empty` indiquant l'absence de valeurs par défaut et d'annotations.

name

Le nom du paramètre (chaîne de caractères). Le nom doit être un identifiant Python valide.

CPython génère un nom de paramètre implicite de forme `.0` sur les objets de code utilisés pour implémenter les compréhensions et expressions génératrices.

Modifié dans la version 3.6 : These parameter names are now exposed by this module as names like `implicit0`.

default

La valeur par défaut de ce paramètre. Si le paramètre n'a pas de valeur par défaut, cet attribut vaut `Parameter.empty`.

annotation

L'annotation du paramètre. Si le paramètre n'a pas d'annotation, cet attribut est défini à `Parameter.empty`.

kind

Describes how argument values are bound to the parameter. The possible values are accessible via `Parameter` (like `Parameter.KEYWORD_ONLY`), and support comparison and ordering, in the following order :

Nom	Signification
POSITIONAL_ONLY	La valeur doit être donnée comme argument positionnel. Les arguments exclusivement positionnels sont ceux qui apparaissent avant une entrée `/` (si présente) dans une définition de fonction Python.
POSITIONAL_OR_KEYWORD	La valeur peut être donnée comme argument nommé ou positionnel (ce qui est l'affectation de variable standard pour les fonctions implémentées en Python).
VAR_POSITIONAL	Un *n*-uplet d'arguments positionnels qui ne sont affectés à aucun autre paramètre. Cela correspond à un paramètre `*args` dans une définition de fonction Python.
KEYWORD_ONLY	La valeur doit être donnée comme argument nommé. Les arguments exclusivement nommés sont ceux qui apparaissent après un `*` ou `*args` dans une définition de fonction Python.
VAR_KEYWORD	Un dictionnaire d'arguments nommés qui ne sont liés à aucun autre paramètre. Cela correspond à une expression `**kwargs` dans une définition de fonction Python.

Example : print all keyword-only arguments without default values :

```
>>> def foo(a, b, *, c, d=10):
...     pass
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> sig = signature(foo)
>>> for param in sig.parameters.values():
...     if (param.kind == param.KEYWORD_ONLY and
...                        param.default is param.empty):
...         print('Parameter:', param)
Parameter: c
```

`kind.`**`description`**

Describes an enum value of `Parameter.kind`.

Ajouté dans la version 3.8.

Example : print all descriptions of arguments :

```
>>> def foo(a, b, *, c, d=10):
...     pass

>>> sig = signature(foo)
>>> for param in sig.parameters.values():
...     print(param.kind.description)
positional or keyword
positional or keyword
keyword-only
keyword-only
```

`replace`(**[, name][, kind][, default][, annotation]*)

Create a new `Parameter` instance based on the instance replaced was invoked on. To override a `Parameter` attribute, pass the corresponding argument. To remove a default value or/and an annotation from a `Parameter`, pass `Parameter.empty`.

```
>>> from inspect import Parameter
>>> param = Parameter('foo', Parameter.KEYWORD_ONLY, default=42)
>>> str(param)
'foo=42'

>>> str(param.replace()) # Will create a shallow copy of 'param'
'foo=42'

>>> str(param.replace(default=Parameter.empty, annotation='spam'))
"foo: 'spam'"
```

`Parameter` objects are also supported by the generic function `copy.replace()`.

Modifié dans la version 3.4 : In Python 3.3 `Parameter` objects were allowed to have `name` set to `None` if their `kind` was set to `POSITIONAL_ONLY`. This is no longer permitted.

`class` `inspect.`**`BoundArguments`**

Le résultat d'un appel à `Signature.bind()` ou `Signature.bind_partial()`. Contient la correspondance entre les arguments et les paramètres de la fonction.

`arguments`

Une correspondance mutable entre les noms de paramètres et les valeurs des arguments. Contient seulement les arguments liés explicitement. Les changements dans `arguments` sont reflétés dans `args` et `kwargs`.

Doit être utilisé en combinaison avec `Signature.parameters` pour le traitement d'arguments.

> **Note**
>
> Les arguments pour lesquels `Signature.bind()` ou `Signature.bind_partial()` supposent une valeur par défaut sont ignorés. Au besoin, utiliser `BoundArguments.apply_defaults()` pour les inclure.

Modifié dans la version 3.9 : `arguments` est maintenant de type `dict`. Anciennement, il était de type `collections.OrderedDict`.

args

Un *n*-uplets des valeurs d'arguments positionnels. Évalué dynamiquement à partir de l'attribut `arguments`.

kwargs

A dict of keyword arguments values. Dynamically computed from the `arguments` attribute. Arguments that can be passed positionally are included in `args` instead.

signature

Une référence à l'objet `Signature` de la classe parente.

apply_defaults()

Assigne les valeurs par défaut pour les arguments manquants.

Pour les arguments positionnels variables (`*args`) la valeur par défaut est un *n*-uplet vide.

Pour les arguments nommés variables (`**kwargs`) la valeur par défaut est un dictionnaire vide.

```
>>> def foo(a, b='ham', *args): pass
>>> ba = inspect.signature(foo).bind('spam')
>>> ba.apply_defaults()
>>> ba.arguments
{'a': 'spam', 'b': 'ham', 'args': ()}
```

Ajouté dans la version 3.5.

The `args` and `kwargs` properties can be used to invoke functions :

```
def test(a, *, b):
    ...

sig = signature(test)
ba = sig.bind(10, b=20)
test(*ba.args, **ba.kwargs)
```

> **Voir aussi**
>
> **PEP 362 — *Function Signature Object.***
> La spécification détaillée, détails d'implémentation et exemples.

30.14.4 Classes et fonctions

inspect.**getclasstree**(*classes*, *unique=False*)

Organise la liste de classes donnée en une hiérarchie de listes imbriquées. Lorsqu'une liste imbriquée apparaît, elle contient les classes dérivées de la classe dont l'entrée précède immédiatement cette liste. Chaque entrée est une paire contenant la classe et un *n*-uplet de ses classes de base. Si l'argument *unique* est vrai, exactement une entrée apparaît dans la structure renvoyée pour chaque classe de la liste donnée en argument. Autrement, les classes utilisant l'héritage multiple et ses descendantes apparaissent plusieurs fois.

`inspect.getfullargspec(func)`

Obtenir les noms et les valeurs par défaut des paramètres de fonction Python. Un *n-uplet nommé* est renvoyé :

```
FullArgSpec(args, varargs, varkw, defaults, kwonlyargs, kwonlydefaults,
annotations)
```

args est une liste de noms de paramètres positionnels. *varargs* est le nom du paramètre `*` ou `None` si les arguments positionnels arbitraires ne sont pas acceptés. *varkw* est le nom du paramètre `**`, ou `None` si les paramètres arbitraires nommés ne sont pas acceptés. *defaults* est un *n*-uplet des valeurs des arguments par défaut correspondant aux derniers *n* arguments positionnels, ou `None` si aucun défaut n'est défini. *kwonlyargs* est une liste d'arguments exclusivement nommés, ordonnée selon l'ordre de déclaration. *kwonlydefaults* est un dictionnaire de correspondance entre les noms de paramètres de *kwonlyargs* et les valeurs par défaut utilisées si aucun argument n'est donné. *annotations* est un dictionnaire de correspondance entre les noms de paramètres et les annotations. La clef spéciale `"return"` est utilisée pour obtenir la valeur de retour de l'annotation la fonction (si elle existe).

Notez que `signature()` et *Signature Object* fournissent l'API recommandé pour l'introspection d'appelables et prennent en charge des comportements additionnels (comme les arguments exclusivement positionnels) qui sont parfois rencontrés dans les APIs des modules d'extension. Cette fonction et maintenue principalement pour maintenir la compatibilité avec l'API du module `inspect` de Python 2.

Modifié dans la version 3.4 : cette fonction est maintenant basée sur `signature()`, mais ignore toujours les attributs `_wrapped_` et inclue le premier paramètre (déjà lié) dans la signature des méthodes liées.

Modifié dans la version 3.6 : cette méthode fût précédemment documentée comme dépréciée en faveur de `signature()` en Python 3.5, mais cette décision fût renversée afin de ré-introduire une interface standard clairement supportée pour du code unifié Python 2/3 migrant pour se départir de l'API `getargspec()`.

Modifié dans la version 3.7 : Python ne garantit explicitement la préservation de l'ordre des paramètres exclusivement nommés que depuis la version 3.7, bien qu'en pratique cet ordre a toujours été préservé sous Python 3.

`inspect.getargvalues(frame)`

Obtenir de l'information sur les arguments passés à un cadre particulier. Un *n-uplet nommé* `ArgInfo(args, varargs, keywords, locals)` est renvoyé. *args* est une liste de noms d'arguments. *varags* et *keywords* sont les noms des arguments `*` et `**` ou `None`. *locals* est le dictionnaire des variables locales du cadre courant.

> **Note**
>
> Cette fonction fût dépréciée par inadvertance en Python 3.5.

`inspect.formatargvalues(args[, varargs, varkw, locals, formatarg, formatvarargs, formatvarkw, formatvalue])`

Formatage lisible des quatre valeurs renvoyées par `getargvalues()`. Les arguments format* sont des fonctions de formatage optionnelles qui sont appelées pour convertir les noms et valeurs en chaînes de caractères.

> **Note**
>
> Cette fonction fût dépréciée par inadvertance en Python 3.5.

`inspect.getmro(cls)`

Renvoie un *n*-uplet des classes de base de *cls*, incluant *cls*, selon le l'ordre de résolution des méthodes (*method resolution order*). Aucune classe n'apparaît plus d'une fois dans ce *n*-uplet. Notez que le l'ordre de résolution des méthodes dépend du type de *cls*. Exception faite d'une utilisation très particulière de la métaclasse définie par l'usager, *cls* sera toujours le premier élément du tuple.

`inspect.getcallargs(func, /, *args, **kwds)`

Bind the *args* and *kwds* to the argument names of the Python function or method *func*, as if it was called with them. For bound methods, bind also the first argument (typically named `self`) to the associated instance. A

dict is returned, mapping the argument names (including the names of the `*` and `**` arguments, if any) to their values from *args* and *kwds*. In case of invoking *func* incorrectly, i.e. whenever `func(*args, **kwds)` would raise an exception because of incompatible signature, an exception of the same type and the same or similar message is raised. For example :

```
>>> from inspect import getcallargs
>>> def f(a, b=1, *pos, **named):
...     pass
...
>>> getcallargs(f, 1, 2, 3) == {'a': 1, 'named': {}, 'b': 2, 'pos': (3,)}
True
>>> getcallargs(f, a=2, x=4) == {'a': 2, 'named': {'x': 4}, 'b': 1, 'pos': ()}
True
>>> getcallargs(f)
Traceback (most recent call last):
...
TypeError: f() missing 1 required positional argument: 'a'
```

Ajouté dans la version 3.2.

Obsolète depuis la version 3.5 : utilisez plutôt `Signature.bind()` et `Signature.bind_partial()`.

`inspect.`**`getclosurevars`**(*func*)

Obtenir la correspondance de nom de références externes d'une fonction ou méthode Python *func* avec leur valeur courante. Un *n-uplet nommé* `ClosureVars(nonlocals, globals, builtins, unbound)` est renvoyé. *nonlocals* établit une correspondance des noms capturés par les fermetures lexicales, *globals* au module de la fonction et *builtins* aux noms natifs visibles depuis le corps de la fonction. *unbound* est un ensemble de noms référencés dans la fonction ne trouvant aucune résolution dans les valeurs globales du module courant ni à travers le lexique natif.

`TypeError` est levée si *func* n'est ni une fonction ni une méthode Python.

Ajouté dans la version 3.3.

`inspect.`**`unwrap`**(*func*, *, *stop=None*)

Obtenir l'objet englobé par *func*. Suit une chaîne d'attributs de `__wrapped__`, renvoyant le dernier objet de la chaîne.

stop est une fonction de *callback* optionnelle qui accepte un objet de la chaîne englobée comme son seul argument et permet d'arrêter le déballage avant la fin si la fonction de *callback* renvoie une valeur vraie. Si la fonction *callback* ne renvoie jamais une valeur vraie, le dernier objet de la chaîne est renvoyé comme d'habitude. Par exemple, `signature()` utilise ceci pour arrêter le déballage si un objet dans la chaîne définit un attribut `__signature__`.

`ValueError` est levée si un cycle est rencontré.

Ajouté dans la version 3.4.

`inspect.`**`get_annotations`**(*obj*, *, *globals=None*, *locals=None*, *eval_str=False*, *format=annotationlib.Format.VALUE*)

Compute the annotations dict for an object.

This is an alias for `annotationlib.get_annotations()` ; see the documentation of that function for more information.

Ajouté dans la version 3.10.

Modifié dans la version 3.14 : This function is now an alias for `annotationlib.get_annotations()`. Calling it as `inspect.get_annotations` will continue to work.

30.14.5 La pile d'interpréteur

Some of the following functions return `FrameInfo` objects. For backwards compatibility these objects allow tuple-like operations on all attributes except `positions`. This behavior is considered deprecated and may be removed in the future.

class `inspect.`**`FrameInfo`**

`frame`

The frame object that the record corresponds to.

`filename`

The file name associated with the code being executed by the frame this record corresponds to.

`lineno`

The line number of the current line associated with the code being executed by the frame this record corresponds to.

`function`

The function name that is being executed by the frame this record corresponds to.

`code_context`

A list of lines of context from the source code that's being executed by the frame this record corresponds to.

`index`

The index of the current line being executed in the `code_context` list.

`positions`

A `dis.Positions` object containing the start line number, end line number, start column offset, and end column offset associated with the instruction being executed by the frame this record corresponds to.

Modifié dans la version 3.5 : Return a *named tuple* instead of a `tuple`.

Modifié dans la version 3.11 : `FrameInfo` is now a class instance (that is backwards compatible with the previous *named tuple*).

class `inspect.`**`Traceback`**

`filename`

The file name associated with the code being executed by the frame this traceback corresponds to.

`lineno`

The line number of the current line associated with the code being executed by the frame this traceback corresponds to.

`function`

The function name that is being executed by the frame this traceback corresponds to.

`code_context`

A list of lines of context from the source code that's being executed by the frame this traceback corresponds to.

`index`

The index of the current line being executed in the `code_context` list.

`positions`

A `dis.Positions` object containing the start line number, end line number, start column offset, and end column offset associated with the instruction being executed by the frame this traceback corresponds to.

Modifié dans la version 3.11 : `Traceback` is now a class instance (that is backwards compatible with the previous *named tuple*).

Note

Conserver des références à des objets cadre, tel que trouvé dans le 1er élément des enregistrements de cadres que ces fonctions renvoient, peut créer des cycles de référence dans votre programme. Lorsqu'un cycle de référence a été créé, la durée de vie de tous les objets pouvant être accessibles par ces objets peut être grandement augmentée, même si le détecteur de cycle (optionnel) de Python est activé. Si ces cycles doivent être créés, il est important de s'assurer qu'ils soient explicitement brisés pour éviter la destruction reportée des objets et l'augmentation résultante de l'utilisation de la mémoire.

Bien que le détecteur trouvera ces cycles, la destruction des cadres (et variables locales) peut être fait de manière déterministe en enlevant les cycles dans une clause `finally`. Cela est important si le détecteur de cycle a été désactivé lors de la compilation du code Python ou en utilisant `gc.disable()`. Par exemple :

```
def handle_stackframe_without_leak():
    frame = inspect.currentframe()
    try:
        # do something with the frame
    finally:
        del frame
```

Pour conserver ces cadres (par exemple pour imprimer une trace d'exécution plus tard), vous pouvez briser ces cycles de référence en utilisant la méthode `frame.clear()`.

L'argument optionnel *context* supporté par la plupart de ces méthodes spécifie le nombre de lignes de contexte à renvoyer, qui sera centré autour de la ligne courante.

`inspect.`**`getframeinfo`**(*frame*, *context=1*)

Get information about a frame or traceback object. A `Traceback` object is returned.

Modifié dans la version 3.11 : A `Traceback` object is returned instead of a named tuple.

`inspect.`**`getouterframes`**(*frame*, *context=1*)

Get a list of `FrameInfo` objects for a frame and all outer frames. These frames represent the calls that lead to the creation of *frame*. The first entry in the returned list represents *frame* ; the last entry represents the outermost call on *frame*'s stack.

Modifié dans la version 3.5 : une liste de *n-uplet nommé* `FrameInfo(frame, filename, lineno, function, code_context, index)` est renvoyée.

Modifié dans la version 3.11 : A list of `FrameInfo` objects is returned.

`inspect.`**`getinnerframes`**(*traceback*, *context=1*)

Get a list of `FrameInfo` objects for a traceback's frame and all inner frames. These frames represent calls made as a consequence of *frame*. The first entry in the list represents *traceback* ; the last entry represents where the exception was raised.

Modifié dans la version 3.5 : une liste de *n-uplet nommé* `FrameInfo(frame, filename, lineno, function, code_context, index)` est renvoyée.

Modifié dans la version 3.11 : A list of `FrameInfo` objects is returned.

`inspect.`**`currentframe`**()

Renvoie l'objet cadre de la pile pour le cadre de l'appelant.

cette fonction nécessite que l'interpréteur Python implémente une pile de cadres d'appel, ce qui n'est pas forcément le cas de toutes les implémentations Python. Si le code s'exécute dans une implémentation Python n'implémentant pas la pile de cadres, cette fonction renvoie `None`.

`inspect.`**`stack`**(*context=1*)

Return a list of `FrameInfo` objects for the caller's stack. The first entry in the returned list represents the caller ; the last entry represents the outermost call on the stack.

Modifié dans la version 3.5 : une liste de *n-uplet nommé* `FrameInfo(frame, filename, lineno, function, code_context, index)` est renvoyée.

Modifié dans la version 3.11 : A list of `FrameInfo` objects is returned.

`inspect.trace(context=1)`

Return a list of `FrameInfo` objects for the stack between the current frame and the frame in which an exception currently being handled was raised in. The first entry in the list represents the caller ; the last entry represents where the exception was raised.

Modifié dans la version 3.5 : une liste de *n-uplet nommé* `FrameInfo(frame, filename, lineno, function, code_context, index)` est renvoyée.

Modifié dans la version 3.11 : A list of `FrameInfo` objects is returned.

30.14.6 Recherche dynamique d'attributs

Both `getattr()` and `hasattr()` can trigger code execution when fetching or checking for the existence of attributes. Descriptors, like properties, will be invoked and `__getattr__()` and `__getattribute__()` may be called.

Pour les cas où une introspection passive est désirée, comme des outils de documentation, cela peut être inconvenable. `getattr_static()` possède la même signature que `getattr()`, mais évite l'exécution de code lors de la recherche d'attributs.

`inspect.getattr_static(obj, attr, default=None)`

Retrieve attributes without triggering dynamic lookup via the descriptor protocol, `__getattr__()` or `__getattribute__()`.

Note : cette fonction ne retrouve pas forcément tous les attributs renvoyés par *getattr* (tel que des attributs générés dynamiquement) et pourrait retrouver des attributs non retrouvés par *getattr* (tel que des descripteurs qui lèvent un *AttributeError*). Elle peut aussi donner des objets descripteurs, plutôt que des membres d'instance.

Si l'instance `__dict__` est cachée (*shadowed*) par un autre membre (par exemple une propriété), cette fonction ne pourra pas retrouver les membres de l'instance.

Ajouté dans la version 3.2.

`getattr_static()` ne résout pas les descripteurs, par exemple *slot descriptors* ou les descripteurs *getset* des objets implémentés en C. Le descripteur de l'objet est renvoyé plutôt que l'attribut sous-jacent.

Vous pouvez gérer ces cas à l'aide du code suivant. Notez que pour des descripteurs arbitraires *getset* les invoquant, de l'exécution de code pourrait être lancée :

```
# example code for resolving the builtin descriptor types
class _foo:
    __slots__ = ['foo']

slot_descriptor = type(_foo.foo)
getset_descriptor = type(type(open(__file__)).name)
wrapper_descriptor = type(str.__dict__['__add__'])
descriptor_types = (slot_descriptor, getset_descriptor, wrapper_descriptor)

result = getattr_static(some_object, 'foo')
if type(result) in descriptor_types:
    try:
        result = result.__get__()
    except AttributeError:
        # descriptors can raise AttributeError to
        # indicate there is no underlying value
        # in which case the descriptor itself will
        # have to do
        pass
```

30.14.7 Current State of Generators, Coroutines, and Asynchronous Generators

Lors de l'implémentation d'ordonnanceur de coroutine et autres utilisations avancées de générateurs, il est utile de déterminer si un générateur est présentement en cours d'exécution, en attente de démarrage ou de continuation, ou s'il a déjà terminé. `getgeneratorstate()` permet d'aisément déterminer l'état courant d'un générateur.

`inspect.`**`getgeneratorstate`**(*generator*)

Donne l'état courant d'un générateur-itérateur.

Les états possibles :
— GEN_CREATED : En attente de démarrage d'exécution.
— GEN_RUNNING : En cours d'exécution par l'interpréteur.
— GEN_SUSPENDED : Présentement suspendu à une expression *yield*.
— GEN_CLOSED : L'exécution est complétée.
Ajouté dans la version 3.2.

`inspect.`**`getcoroutinestate`**(*coroutine*)

Donne l'état courant d'un objet coroutine. L'utilisation est prévue avec des objets coroutine créés avec des fonctions `async def`, mais accepte n'importe quel objet semblable à une coroutine qui définisse des attributs `cr_running` `cr_frame`.

Les états possibles :
— CORO_CREATED : En attente de démarrage d'exécution.
— CORO_RUNNING : En cours d'exécution par l'interpréteur.
— CORO_SUSPENDED : Suspendu à une expression *await*.
— CORO_CLOSED : Exécution complétée.
Ajouté dans la version 3.5.

`inspect.`**`getasyncgenstate`**(*agen*)

Get current state of an asynchronous generator object. The function is intended to be used with asynchronous iterator objects created by `async def` functions which use the `yield` statement, but will accept any asynchronous generator-like object that has `ag_running` and `ag_frame` attributes.

Les états possibles :
— AGEN_CREATED : Waiting to start execution.
— AGEN_RUNNING : Currently being executed by the interpreter.
— AGEN_SUSPENDED : Currently suspended at a yield expression.
— AGEN_CLOSED : Execution has completed.
Ajouté dans la version 3.12.

L'état courant interne du générateur peut aussi être vérifié. Ceci est utile pour des tests, afin d'assurer que l'état interne est mise à jour tel que prévu :

`inspect.`**`getgeneratorlocals`**(*generator*)

Donne une correspondance de variables locales dans *generator* avec les valeurs courantes. Un dictionnaire est renvoyé, établissant la correspondance entre variables et valeurs. Ceci est équivalent à appeler `locals()` dans le corps du générateur et les mêmes restrictions s'appliquent.

Si *generator* est un *générateur* sans cadre courant associé, alors un dictionnaire vide est renvoyé. `TypeError` est levé sir *generator* n'est pas un objet générateur Python.

Cette fonction repose sur l'exposition d'une pile de cadres Python par le générateur pour l'introspection, ce qui n'est pas garanti dans toutes les implémentations Python. Dans un tel cas, cette fonction renvoie toujours un dictionnaire vide.

Ajouté dans la version 3.3.

`inspect.`**`getcoroutinelocals`**(*coroutine*)

Cette fonction est semblable à `getgeneratorlocals()`, mais fonctionne pour des objets coroutines créés avec des fonctions `async def`.

Ajouté dans la version 3.5.

`inspect.getasyncgenlocals(agen)`

This function is analogous to `getgeneratorlocals()`, but works for asynchronous generator objects created by `async def` functions which use the `yield` statement.

Ajouté dans la version 3.12.

30.14.8 Bit d'option des objets code

Python code objects have a `co_flags` attribute, which is a bitmap of the following flags :

`inspect.CO_OPTIMIZED`

L'objet code est optimisé, utilisant des variables locales rapides.

`inspect.CO_NEWLOCALS`

If set, a new dict will be created for the frame's `f_locals` when the code object is executed.

`inspect.CO_VARARGS`

L'objet code possède un paramètre de variable positionnel (`*args` ou semblable).

`inspect.CO_VARKEYWORDS`

L'objet code possède une paramètre par mot-clé variable (`**kwargs` ou semblable).

`inspect.CO_NESTED`

L'option est définie lorsque le code est une fonction imbriquée.

`inspect.CO_GENERATOR`

L'option est définie lorsque l'objet code est une fonction génératrice, c'est-à-dire qu'un objet générateur est renvoyé lorsque le code objet est exécuté.

`inspect.CO_COROUTINE`

L'option est définie lorsque le code objet est une fonction coroutine. Lors que le code objet est exécuté, il renvoie un objet coroutine. Voir **PEP 492** pour plus de détails.

Ajouté dans la version 3.5.

`inspect.CO_ITERABLE_COROUTINE`

L'option est utilisée pour transformer un générateur en une coroutine basée sur un générateur. Les objets générateurs avec cette option peuvent être utilisés dans une expression `await` et un objet coroutine `yield from`. Voir **PEP 492** pour plus de détails.

Ajouté dans la version 3.5.

`inspect.CO_ASYNC_GENERATOR`

L'option est définie lorsque l'objet code est une fonction génératrice asynchrone. Lorsque l'objet code est exécuté, il renvoie un objet générateur asynchrone. Voir **PEP 525** pour les détails.

Ajouté dans la version 3.6.

`inspect.CO_HAS_DOCSTRING`

The flag is set when there is a docstring for the code object in the source code. If set, it will be the first item in `co_consts`.

Ajouté dans la version 3.14.

`inspect.CO_METHOD`

The flag is set when the code object is a function defined in class scope.

Ajouté dans la version 3.14.

Note

Ces options sont spécifiques à CPython et pourraient ne pas être définies dans d'autres implémentations Python. De plus, les options constituent des détails d'implémentation et pourraient être enlevées ou dépréciées dans des

versions ultérieures. Il est recommandé d'utiliser les APIs publiques de *inspect* pour tous les besoins d'introspection.

30.14.9 Buffer flags

class inspect.**BufferFlags**

This is an *enum.IntFlag* that represents the flags that can be passed to the `__buffer__()` method of objects implementing the buffer protocol.

The meaning of the flags is explained at buffer-request-types.

SIMPLE

WRITABLE

FORMAT

ND

STRIDES

C_CONTIGUOUS

F_CONTIGUOUS

ANY_CONTIGUOUS

INDIRECT

CONTIG

CONTIG_RO

STRIDED

STRIDED_RO

RECORDS

RECORDS_RO

FULL

FULL_RO

READ

WRITE

Ajouté dans la version 3.12.

30.14.10 Interface en ligne de commande

Le module *inspect* fournit également des capacités d'introspection de base en ligne de commande.

Par défaut, il accepte le nom d'un module et affiche la source de ce module. Une classe ou fonction à l'intérieur du module peut également être affichée en ajoutant un " :" et le nom qualifié de l'objet désiré.

--details

Affiche de l'information à propos de l'objet spécifié plutôt que du code source.

30.15 annotationlib --- Functionality for introspecting annotations

Source code : Lib/annotationlib.py

The annotationlib module provides tools for introspecting *annotations* on modules, classes, and functions.

Annotations are lazily evaluated and often contain forward references to objects that are not yet defined when the annotation is created. This module provides a set of low-level tools that can be used to retrieve annotations in a reliable way, even in the presence of forward references and other edge cases.

This module supports retrieving annotations in three main formats (see *Format*), each of which works best for different use cases :

- *VALUE* evaluates the annotations and returns their value. This is most straightforward to work with, but it may raise errors, for example if the annotations contain references to undefined names.
- *FORWARDREF* returns *ForwardRef* objects for annotations that cannot be resolved, allowing you to inspect the annotations without evaluating them. This is useful when you need to work with annotations that may contain unresolved forward references.
- *STRING* returns the annotations as a string, similar to how it would appear in the source file. This is useful for documentation generators that want to display annotations in a readable way.

The *get_annotations()* function is the main entry point for retrieving annotations. Given a function, class, or module, it returns an annotations dictionary in the requested format. This module also provides functionality for working directly with the *annotate function* that is used to evaluate annotations, such as *get_annotate_function()* and *call_annotate_function()*, as well as the *call_evaluate_function()* function for working with *evaluate functions*.

Voir aussi

PEP 649 proposed the current model for how annotations work in Python.

PEP 749 expanded on various aspects of **PEP 649** and introduced the annotationlib module.

annotations-howto provides best practices for working with annotations.

typing-extensions provides a backport of *get_annotations()* that works on earlier versions of Python.

30.15.1 Annotation semantics

The way annotations are evaluated has changed over the history of Python 3, and currently still depends on a future import. There have been execution models for annotations :

- *Stock semantics* (default in Python 3.0 through 3.13 ; see **PEP 3107** and **PEP 526**) : Annotations are evaluated eagerly, as they are encountered in the source code.
- *Stringified annotations* (used with from __future__ import annotations in Python 3.7 and newer ; see **PEP 563**) : Annotations are stored as strings only.
- *Deferred evaluation* (default in Python 3.14 and newer ; see **PEP 649** and **PEP 749**) : Annotations are evaluated lazily, only when they are accessed.

As an example, consider the following program :

```
def func(a: Cls) -> None:
    print(a)

class Cls: pass

print(func.__annotations__)
```

This will behave as follows :

- Under stock semantics (Python 3.13 and earlier), it will throw a *NameError* at the line where func is defined, because Cls is an undefined name at that point.
- Under stringified annotations (if from __future__ import annotations is used), it will print {'a': 'Cls', 'return': 'None'}.

— Under deferred evaluation (Python 3.14 and later), it will print `{'a': <class 'Cls'>, 'return': None}`.

Stock semantics were used when function annotations were first introduced in Python 3.0 (by **PEP 3107**) because this was the simplest, most obvious way to implement annotations. The same execution model was used when variable annotations were introduced in Python 3.6 (by **PEP 526**). However, stock semantics caused problems when using annotations as type hints, such as a need to refer to names that are not yet defined when the annotation is encountered. In addition, there were performance problems with executing annotations at module import time. Therefore, in Python 3.7, **PEP 563** introduced the ability to store annotations as strings using the `from __future__ import annotations` syntax. The plan at the time was to eventually make this behavior the default, but a problem appeared : stringified annotations are more difficult to process for those who introspect annotations at runtime. An alternative proposal, **PEP 649**, introduced the third execution model, deferred evaluation, and was implemented in Python 3.14. Stringified annotations are still used if `from __future__ import annotations` is present, but this behavior will eventually be removed.

30.15.2 Classes

class annotationlib.**Format**

An *IntEnum* describing the formats in which annotations can be returned. Members of the enum, or their equivalent integer values, can be passed to *get_annotations()* and other functions in this module, as well as to `__annotate__` functions.

VALUE = 1

Values are the result of evaluating the annotation expressions.

FORWARDREF = 2

Values are real annotation values (as per *Format.VALUE* format) for defined values, and *ForwardRef* proxies for undefined values. Real objects may contain references to, *ForwardRef* proxy objects.

STRING = 3

Values are the text string of the annotation as it appears in the source code, up to modifications including, but not restricted to, whitespace normalizations and constant values optimizations.

The exact values of these strings may change in future versions of Python.

VALUE_WITH_FAKE_GLOBALS = 4

Special value used to signal that an annotate function is being evaluated in a special environment with fake globals. When passed this value, annotate functions should either return the same value as for the *Format.VALUE* format, or raise *NotImplementedError* to signal that they do not support execution in this environment. This format is only used internally and should not be passed to the functions in this module.

Ajouté dans la version 3.14.

class annotationlib.**ForwardRef**

A proxy object for forward references in annotations.

Instances of this class are returned when the *FORWARDREF* format is used and annotations contain a name that cannot be resolved. This can happen when a forward reference is used in an annotation, such as when a class is referenced before it is defined.

__forward_arg__

A string containing the code that was evaluated to produce the *ForwardRef*. The string may not be exactly equivalent to the original source.

evaluate(*, *globals=None*, *locals=None*, *type_params=None*, *owner=None*)

Evaluate the forward reference, returning its value.

This may throw an exception, such as *NameError*, if the forward reference refers to names that do not exist. The arguments to this method can be used to provide bindings for names that would otherwise be undefined.

ForwardRef instances returned by *get_annotations()* retain references to information about the scope they originated from, so calling this method with no further arguments may be sufficient to evaluate

such objects. `ForwardRef` instances created by other means may not have any information about their scope, so passing arguments to this method may be necessary to evaluate them successfully.

globals and *locals* are passed to `eval()`, representing the global and local namespaces in which the name is evaluated. *type_params*, if given, must be a tuple of type parameters that are in scope while the forward reference is being evaluated. *owner* is the object that owns the annotation from which the forward reference derives, usually a function, class, or module.

> **Important**
>
> Once a `ForwardRef` instance has been evaluated, it caches the evaluated value, and future calls to `evaluate()` will return the cached value, regardless of the parameters passed in.

Ajouté dans la version 3.14.

30.15.3 Functions

`annotationlib.`**`annotations_to_string`**(*annotations*)

Convert an annotations dict containing runtime values to a dict containing only strings. If the values are not already strings, they are converted using `value_to_string()`. This is meant as a helper for user-provided annotate functions that support the `STRING` format but do not have access to the code creating the annotations.

For example, this is used to implement the `STRING` for `typing.TypedDict` classes created through the functional syntax :

```
>>> from typing import TypedDict
>>> Movie = TypedDict("movie", {"name": str, "year": int})
>>> get_annotations(Movie, format=Format.STRING)
{'name': 'str', 'year': 'int'}
```

Ajouté dans la version 3.14.

`annotationlib.`**`call_annotate_function`**(*annotate*, *format*, *, *owner=None*)

Call the *annotate function* *annotate* with the given *format*, a member of the `Format` enum, and return the annotations dictionary produced by the function.

This helper function is required because annotate functions generated by the compiler for functions, classes, and modules only support the `VALUE` format when called directly. To support other formats, this function calls the annotate function in a special environment that allows it to produce annotations in the other formats. This is a useful building block when implementing functionality that needs to partially evaluate annotations while a class is being constructed.

owner is the object that owns the annotation function, usually a function, class, or module. If provided, it is used in the `FORWARDREF` format to produce a `ForwardRef` object that carries more information.

> **Voir aussi**
>
> **PEP 649** contains an explanation of the implementation technique used by this function.

Ajouté dans la version 3.14.

`annotationlib.`**`call_evaluate_function`**(*evaluate*, *format*, *, *owner=None*)

Call the *evaluate function* *evaluate* with the given *format*, a member of the `Format` enum, and return the value produced by the function. This is similar to `call_annotate_function()`, but the latter always returns a dictionary mapping strings to annotations, while this function returns a single value.

This is intended for use with the evaluate functions generated for lazily evaluated elements related to type aliases and type parameters :

— `typing.TypeAliasType.evaluate_value()`, the value of type aliases

— *typing.TypeVar.evaluate_bound()*, the bound of type variables
— *typing.TypeVar.evaluate_constraints()*, the constraints of type variables
— *typing.TypeVar.evaluate_default()*, the default value of type variables
— *typing.ParamSpec.evaluate_default()*, the default value of parameter specifications
— *typing.TypeVarTuple.evaluate_default()*, the default value of type variable tuples

owner is the object that owns the evaluate function, such as the type alias or type variable object.

format can be used to control the format in which the value is returned :

```
>>> type Alias = undefined
>>> call_evaluate_function(Alias.evaluate_value, Format.VALUE)
Traceback (most recent call last):
...
NameError: name 'undefined' is not defined
>>> call_evaluate_function(Alias.evaluate_value, Format.FORWARDREF)
ForwardRef('undefined')
>>> call_evaluate_function(Alias.evaluate_value, Format.STRING)
'undefined'
```

Ajouté dans la version 3.14.

annotationlib.**get_annotate_function**(*obj*)

Retrieve the *annotate function* for *obj*. Return `None` if *obj* does not have an annotate function.

This is usually equivalent to accessing the `__annotate__` attribute of *obj*, but direct access to the attribute may return the wrong object in certain situations involving metaclasses. This function should be used instead of accessing the attribute directly.

Ajouté dans la version 3.14.

annotationlib.**get_annotations**(*obj*, *, *globals=None*, *locals=None*, *eval_str=False*, *format=Format.VALUE*)

Compute the annotations dict for an object.

obj may be a callable, class, module, or other object with `__annotate__` and `__annotations__` attributes. Passing in an object of any other type raises *TypeError*.

The *format* parameter controls the format in which annotations are returned, and must be a member of the *Format* enum or its integer equivalent.

Returns a dict. `get_annotations()` returns a new dict every time it's called ; calling it twice on the same object will return two different but equivalent dicts.

This function handles several details for you :
— If *eval_str* is true, values of type `str` will be un-stringized using *eval()*. This is intended for use with stringized annotations (`from __future__ import annotations`). It is an error to set *eval_str* to true with formats other than *Format.VALUE*.
— If *obj* doesn't have an annotations dict, returns an empty dict. (Functions and methods always have an annotations dict ; classes, modules, and other types of callables may not.)
— Ignores inherited annotations on classes, as well as annotations on metaclasses. If a class doesn't have its own annotations dict, returns an empty dict.
— All accesses to object members and dict values are done using `getattr()` and `dict.get()` for safety.

eval_str controls whether or not values of type `str` are replaced with the result of calling *eval()* on those values :
— If eval_str is true, *eval()* is called on values of type `str`. (Note that `get_annotations()` doesn't catch exceptions ; if *eval()* raises an exception, it will unwind the stack past the `get_annotations()` call.)
— If *eval_str* is false (the default), values of type `str` are unchanged.

globals and *locals* are passed in to *eval()* ; see the documentation for *eval()* for more information. If *globals* or *locals* is `None`, this function may replace that value with a context-specific default, contingent on `type(obj)` :
— If *obj* is a module, *globals* defaults to `obj.__dict__`.
— If *obj* is a class, *globals* defaults to `sys.modules[obj.__module__].__dict__` and *locals* defaults to

the *obj* class namespace.

— If *obj* is a callable, *globals* defaults to `obj.__globals__`, although if *obj* is a wrapped function (using `functools.update_wrapper()`) or a `functools.partial` object, it is unwrapped until a non-wrapped function is found.

Calling `get_annotations()` is best practice for accessing the annotations dict of any object. See annotations-howto for more information on annotations best practices.

```
>>> def f(a: int, b: str) -> float:
...     pass
>>> get_annotations(f)
{'a': <class 'int'>, 'b': <class 'str'>, 'return': <class 'float'>}
```

Ajouté dans la version 3.14.

`annotationlib.value_to_string(value)`

Convert an arbitrary Python value to a format suitable for use by the `STRING` format. This calls `repr()` for most objects, but has special handling for some objects, such as type objects.

This is meant as a helper for user-provided annotate functions that support the `STRING` format but do not have access to the code creating the annotations. It can also be used to provide a user-friendly string representation for other objects that contain values that are commonly encountered in annotations.

Ajouté dans la version 3.14.

30.16 `site` --- Site-specific configuration hook

Code source : Lib/site.py

This module is automatically imported during initialization. The automatic import can be suppressed using the interpreter's `-S` option.

Importing this module normally appends site-specific paths to the module search path and adds *callables*, including `help()` to the built-in namespace. However, Python startup option `-S` blocks this and this module can be safely imported with no automatic modifications to the module search path or additions to the builtins. To explicitly trigger the usual site-specific additions, call the `main()` function.

Modifié dans la version 3.3 : Importing the module used to trigger paths manipulation even when using `-S`.

It starts by constructing up to four directories from a head and a tail part. For the head part, it uses `sys.prefix` and `sys.exec_prefix`; empty heads are skipped. For the tail part, it uses the empty string and then `lib/site-packages` (on Windows) or `lib/pythonX.Y[t]/site-packages` (on Unix and macOS). (The optional suffix "t" indicates the *free threading* build, and is appended if `"t"` is present in the `sys.abiflags` constant.) For each of the distinct head-tail combinations, it sees if it refers to an existing directory, and if so, adds it to `sys.path` and also inspects the newly added path for configuration files.

Modifié dans la version 3.5 : Support for the "site-python" directory has been removed.

Modifié dans la version 3.13 : On Unix, *Free threading* Python installations are identified by the "t" suffix in the version-specific directory name, such as `lib/python3.13t/`.

Modifié dans la version 3.14 : `site` is no longer responsible for updating `sys.prefix` and `sys.exec_prefix` on *Virtual Environments*. This is now done during the *path initialization*. As a result, under *Virtual Environments*, `sys.prefix` and `sys.exec_prefix` no longer depend on the `site` initialization, and are therefore unaffected by `-S`. When running under a *virtual environment*, the `pyvenv.cfg` file in `sys.prefix` is checked for site-specific configurations. If the `include-system-site-packages` key exists and is set to `true` (case-insensitive), the system-level prefixes will be searched for site-packages, otherwise they won't.

A path configuration file is a file whose name has the form `name.pth` and exists in one of the four directories mentioned above; its contents are additional items (one per line) to be added to `sys.path`. Non-existing items are never added to `sys.path`, and no check is made that the item refers to a directory rather than a file. No item is

added to `sys.path` more than once. Blank lines and lines beginning with `#` are skipped. Lines starting with `import` (followed by space or tab) are executed.

> **Note**
>
> An executable line in a `.pth` file is run at every Python startup, regardless of whether a particular module is actually going to be used. Its impact should thus be kept to a minimum. The primary intended purpose of executable lines is to make the corresponding module(s) importable (load 3rd-party import hooks, adjust `PATH` etc). Any other initialization is supposed to be done upon a module's actual import, if and when it happens. Limiting a code chunk to a single line is a deliberate measure to discourage putting anything more complex here.

Modifié dans la version 3.13 : The `.pth` files are now decoded by UTF-8 at first and then by the *locale encoding* if it fails.

For example, suppose `sys.prefix` and `sys.exec_prefix` are set to `/usr/local`. The Python X.Y library is then installed in `/usr/local/lib/pythonX.Y`. Suppose this has a subdirectory `/usr/local/lib/pythonX.Y/site-packages` with three subsubdirectories, `foo`, `bar` and `spam`, and two path configuration files, `foo.pth` and `bar.pth`. Assume `foo.pth` contains the following :

```
# foo package configuration

foo
bar
bletch
```

and `bar.pth` contains :

```
# bar package configuration

bar
```

Then the following version-specific directories are added to `sys.path`, in this order :

```
/usr/local/lib/pythonX.Y/site-packages/bar
/usr/local/lib/pythonX.Y/site-packages/foo
```

Note that `bletch` is omitted because it doesn't exist ; the `bar` directory precedes the `foo` directory because `bar.pth` comes alphabetically before `foo.pth` ; and `spam` is omitted because it is not mentioned in either path configuration file.

30.16.1 sitecustomize

After these path manipulations, an attempt is made to import a module named *sitecustomize*, which can perform arbitrary site-specific customizations. It is typically created by a system administrator in the site-packages directory. If this import fails with an *ImportError* or its subclass exception, and the exception's *name* attribute equals to `'sitecustomize'`, it is silently ignored. If Python is started without output streams available, as with `pythonw.exe` on Windows (which is used by default to start IDLE), attempted output from *sitecustomize* is ignored. Any other exception causes a silent and perhaps mysterious failure of the process.

30.16.2 usercustomize

After this, an attempt is made to import a module named *usercustomize*, which can perform arbitrary user-specific customizations, if *ENABLE_USER_SITE* is true. This file is intended to be created in the user site-packages directory (see below), which is part of `sys.path` unless disabled by `-s`. If this import fails with an *ImportError* or its subclass exception, and the exception's *name* attribute equals to `'usercustomize'`, it is silently ignored.

Note that for some non-Unix systems, `sys.prefix` and `sys.exec_prefix` are empty, and the path manipulations are skipped ; however the import of *sitecustomize* and *usercustomize* is still attempted.

30.16.3 Readline configuration

On systems that support *readline*, this module will also import and configure the *rlcompleter* module, if Python is started in interactive mode and without the `-S` option. The default behavior is enable tab-completion and to use `~/.python_history` as the history save file. To disable it, delete (or override) the *sys.__interactivehook__* attribute in your *sitecustomize* or *usercustomize* module or your `PYTHONSTARTUP` file.

Modifié dans la version 3.4 : Activation of rlcompleter and history was made automatic.

30.16.4 Module contents

`site.`**`PREFIXES`**

A list of prefixes for site-packages directories.

`site.`**`ENABLE_USER_SITE`**

Flag showing the status of the user site-packages directory. `True` means that it is enabled and was added to `sys.path`. `False` means that it was disabled by user request (with `-s` or `PYTHONNOUSERSITE`). `None` means it was disabled for security reasons (mismatch between user or group id and effective id) or by an administrator.

`site.`**`USER_SITE`**

Path to the user site-packages for the running Python. Can be `None` if *getusersitepackages()* hasn't been called yet. Default value is `~/.local/lib/pythonX.Y[t]/site-packages` for UNIX and non-framework macOS builds, `~/Library/Python/X.Y/lib/python/site-packages` for macOS framework builds, and `%APPDATA%\Python\PythonXY\site-packages` on Windows. The optional "t" indicates the free-threaded build. This directory is a site directory, which means that `.pth` files in it will be processed.

`site.`**`USER_BASE`**

Path to the base directory for the user site-packages. Can be `None` if *getuserbase()* hasn't been called yet. Default value is `~/.local` for UNIX and macOS non-framework builds, `~/Library/Python/X.Y` for macOS framework builds, and `%APPDATA%\Python` for Windows. This value is used to compute the installation directories for scripts, data files, Python modules, etc. for the *user installation scheme*. See also `PYTHONUSERBASE`.

`site.`**`main`**`()`

Adds all the standard site-specific directories to the module search path. This function is called automatically when this module is imported, unless the Python interpreter was started with the `-S` flag.

Modifié dans la version 3.3 : This function used to be called unconditionally.

`site.`**`addsitedir`**`(`*sitedir*, *known_paths=None*`)`

Add a directory to sys.path and process its `.pth` files. Typically used in *sitecustomize* or *usercustomize* (see above).

`site.`**`getsitepackages`**`()`

Return a list containing all global site-packages directories.

Ajouté dans la version 3.2.

`site.`**`getuserbase`**`()`

Return the path of the user base directory, *USER_BASE*. If it is not initialized yet, this function will also set it, respecting `PYTHONUSERBASE`.

Ajouté dans la version 3.2.

`site.`**`getusersitepackages`**`()`

Return the path of the user-specific site-packages directory, *USER_SITE*. If it is not initialized yet, this function will also set it, respecting *USER_BASE*. To determine if the user-specific site-packages was added to `sys.path` *ENABLE_USER_SITE* should be used.

Ajouté dans la version 3.2.

30.16.5 Interface en ligne de commande

The `site` module also provides a way to get the user directories from the command line :

```
$ python -m site --user-site
/home/user/.local/lib/python3.11/site-packages
```

If it is called without arguments, it will print the contents of `sys.path` on the standard output, followed by the value of `USER_BASE` and whether the directory exists, then the same thing for `USER_SITE`, and finally the value of `ENABLE_USER_SITE`.

--user-base

Print the path to the user base directory.

--user-site

Print the path to the user site-packages directory.

If both options are given, user base and user site will be printed (always in this order), separated by `os.pathsep`.

If any option is given, the script will exit with one of these values : `0` if the user site-packages directory is enabled, `1` if it was disabled by the user, `2` if it is disabled for security reasons or by an administrator, and a value greater than 2 if there is an error.

Voir aussi

— **PEP 370** -- Répertoire site-packages propre à l'utilisateur.
— *The initialization of the sys.path module search path* -- The initialization of `sys.path`.

CHAPITRE 31

Interpréteurs Python personnalisés

The modules described in this chapter allow writing interfaces similar to Python's interactive interpreter. If you want a Python interpreter that supports some special feature in addition to the Python language, you should look at the `code` module. (The `codeop` module is lower-level, used to support compiling a possibly incomplete chunk of Python code.)

La liste complète des modules décrits dans ce chapitre est :

31.1 `code` --- Interpreter base classes

Code source : Lib/code.py

The `code` module provides facilities to implement read-eval-print loops in Python. Two classes and convenience functions are included which can be used to build applications which provide an interactive interpreter prompt.

class `code.`**`InteractiveInterpreter`**(*locals=None*)

This class deals with parsing and interpreter state (the user's namespace); it does not deal with input buffering or prompting or input file naming (the filename is always passed in explicitly). The optional *locals* argument specifies a mapping to use as the namespace in which code will be executed; it defaults to a newly created dictionary with key `'__name__'` set to `'__console__'` and key `'__doc__'` set to `None`.

class `code.`**`InteractiveConsole`**(*locals=None*, *filename='<console>'*, *local_exit=False*)

Closely emulate the behavior of the interactive Python interpreter. This class builds on `InteractiveInterpreter` and adds prompting using the familiar `sys.ps1` and `sys.ps2`, and input buffering. If *local_exit* is true, `exit()` and `quit()` in the console will not raise `SystemExit`, but instead return to the calling code.

Modifié dans la version 3.13 : Added *local_exit* parameter.

`code.`**`interact`**(*banner=None*, *readfunc=None*, *local=None*, *exitmsg=None*, *local_exit=False*)

Convenience function to run a read-eval-print loop. This creates a new instance of `InteractiveConsole` and sets *readfunc* to be used as the `InteractiveConsole.raw_input()` method, if provided. If *local* is provided, it is passed to the `InteractiveConsole` constructor for use as the default namespace for the interpreter loop. If *local_exit* is provided, it is passed to the `InteractiveConsole` constructor. The `interact()` method of the instance is then run with *banner* and *exitmsg* passed as the banner and exit message to use, if provided. The console object is discarded after use.

Modifié dans la version 3.6 : Added *exitmsg* parameter.

Modifié dans la version 3.13 : Added *local_exit* parameter.

`code.compile_command(source, filename='<input>', symbol='single')`

This function is useful for programs that want to emulate Python's interpreter main loop (a.k.a. the read-eval-print loop). The tricky part is to determine when the user has entered an incomplete command that can be completed by entering more text (as opposed to a complete command or a syntax error). This function *almost* always makes the same decision as the real interpreter main loop.

source is the source string ; *filename* is the optional filename from which source was read, defaulting to `'<input>'` ; and *symbol* is the optional grammar start symbol, which should be `'single'` (the default), `'eval'` or `'exec'`.

Returns a code object (the same as `compile(source, filename, symbol)`) if the command is complete and valid ; `None` if the command is incomplete ; raises `SyntaxError` if the command is complete and contains a syntax error, or raises `OverflowError` or `ValueError` if the command contains an invalid literal.

31.1.1 Interactive Interpreter Objects

`InteractiveInterpreter.runsource(source, filename='<input>', symbol='single')`

Compile and run some source in the interpreter. Arguments are the same as for `compile_command()` ; the default for *filename* is `'<input>'`, and for *symbol* is `'single'`. One of several things can happen :

— The input is incorrect ; `compile_command()` raised an exception (`SyntaxError` or `OverflowError`). A syntax traceback will be printed by calling the `showsyntaxerror()` method. `runsource()` returns `False`.

— The input is incomplete, and more input is required ; `compile_command()` returned `None`. `runsource()` returns `True`.

— The input is complete ; `compile_command()` returned a code object. The code is executed by calling the `runcode()` (which also handles run-time exceptions, except for `SystemExit`). `runsource()` returns `False`.

The return value can be used to decide whether to use `sys.ps1` or `sys.ps2` to prompt the next line.

`InteractiveInterpreter.runcode(code)`

Execute a code object. When an exception occurs, `showtraceback()` is called to display a traceback. All exceptions are caught except `SystemExit`, which is allowed to propagate.

A note about `KeyboardInterrupt` : this exception may occur elsewhere in this code, and may not always be caught. The caller should be prepared to deal with it.

`InteractiveInterpreter.showsyntaxerror(filename=None)`

Display the syntax error that just occurred. This does not display a stack trace because there isn't one for syntax errors. If *filename* is given, it is stuffed into the exception instead of the default filename provided by Python's parser, because it always uses `'<string>'` when reading from a string. The output is written by the `write()` method.

`InteractiveInterpreter.showtraceback()`

Display the exception that just occurred. We remove the first stack item because it is within the interpreter object implementation. The output is written by the `write()` method.

Modifié dans la version 3.5 : The full chained traceback is displayed instead of just the primary traceback.

`InteractiveInterpreter.write(data)`

Write a string to the standard error stream (`sys.stderr`). Derived classes should override this to provide the appropriate output handling as needed.

31.1.2 Interactive Console Objects

The `InteractiveConsole` class is a subclass of `InteractiveInterpreter`, and so offers all the methods of the interpreter objects as well as the following additions.

`InteractiveConsole.interact(banner=None, exitmsg=None)`

Closely emulate the interactive Python console. The optional *banner* argument specify the banner to print before the first interaction ; by default it prints a banner similar to the one printed by the standard Python

interpreter, followed by the class name of the console object in parentheses (so as not to confuse this with the real interpreter -- since it's so close !).

The optional *exitmsg* argument specifies an exit message printed when exiting. Pass the empty string to suppress the exit message. If *exitmsg* is not given or `None`, a default message is printed.

Modifié dans la version 3.4 : To suppress printing any banner, pass an empty string.

Modifié dans la version 3.6 : Print an exit message when exiting.

`InteractiveConsole.push(line)`

Push a line of source text to the interpreter. The line should not have a trailing newline ; it may have internal newlines. The line is appended to a buffer and the interpreter's `runsource()` method is called with the concatenated contents of the buffer as source. If this indicates that the command was executed or invalid, the buffer is reset ; otherwise, the command is incomplete, and the buffer is left as it was after the line was appended. The return value is `True` if more input is required, `False` if the line was dealt with in some way (this is the same as `runsource()`).

`InteractiveConsole.resetbuffer()`

Remove any unhandled source text from the input buffer.

`InteractiveConsole.raw_input(prompt='')`

Write a prompt and read a line. The returned line does not include the trailing newline. When the user enters the EOF key sequence, `EOFError` is raised. The base implementation reads from `sys.stdin` ; a subclass may replace this with a different implementation.

31.2 `codeop` --- Compile Python code

Code source : Lib/codeop.py

Le module `codeop` fournit des outils permettant d'émuler une boucle de lecture-évaluation-affichage (en anglais *read-eval-print-loop* ou REPL), comme dans le module `code`. Par conséquent, ce module n'est pas destiné à être utilisé directement ; pour inclure un REPL dans un programme, il est préférable d'utiliser le module `code`.

Cette tâche se divise en deux parties :

1. Being able to tell if a line of input completes a Python statement : in short, telling whether to print '>>>' or '...' next.
2. Remembering which future statements the user has entered, so subsequent input can be compiled with these in effect.

Le module `codeop` fournit un moyen d'effectuer ces deux parties, individuellement ou simultanément.

Pour ne faire que la première partie :

`codeop.compile_command(source, filename='<input>', symbol='single')`

Tries to compile *source*, which should be a string of Python code and return a code object if *source* is valid Python code. In that case, the filename attribute of the code object will be *filename*, which defaults to `'<input>'`. Returns `None` if *source* is *not* valid Python code, but is a prefix of valid Python code.

En cas de problème avec *source*, une exception est levée ; `SyntaxError` si la syntaxe Python est incorrecte, et `OverflowError` ou `ValueError` si un littéral invalide est rencontré.

The *symbol* argument determines whether *source* is compiled as a statement (`'single'`, the default), as a sequence of *statement* (`'exec'`) or as an *expression* (`'eval'`). Any other value will cause `ValueError` to be raised.

> **Note**
>
> Il est possible (quoique improbable) que l'analyseur s'arrête avant d'atteindre la fin du code source ; dans ce cas, les symboles venant après peuvent être ignorés au lieu de provoquer une erreur. Par exemple, une

barre oblique inverse suivie de deux retours à la ligne peut être suivie par de la mémoire non-initialisée. Ceci sera corrigé quand l'interface de l'analyseur aura été améliorée.

class codeop.**Compile**

Instances of this class have `__call__()` methods identical in signature to the built-in function *compile()*, but with the difference that if the instance compiles program text containing a *__future__* statement, the instance 'remembers' and compiles all subsequent program texts with the statement in force.

class codeop.**CommandCompiler**

Instances of this class have `__call__()` methods identical in signature to *compile_command()* ; the difference is that if the instance compiles program text containing a *__future__* statement, the instance 'remembers' and compiles all subsequent program texts with the statement in force.

CHAPITRE 32

Importer des modules

Les modules décrits dans ce chapitre fournissent de nouveaux moyens d'importer d'autres modules Python, et des *hooks* pour personnaliser le processus d'importation.

La liste complète des modules décrits dans ce chapitre est :

32.1 `zipimport` --- Import modules from Zip archives

Code source : Lib/zipimport.py

Ce module ajoute la possibilité d'importer des modules Python (`*.py`, `*.pyc`) et des paquets depuis des archives au format ZIP. Il n'est généralement pas nécessaire d'utiliser explicitement le module `zipimport` ; il est automatiquement utilisé par le mécanisme intégré de `import` pour les éléments de `sys.path` qui sont des chemins vers des archives ZIP.

En général, `sys.path` est une liste de noms de répertoires sous forme de chaînes. Ce module permet également à un élément de `sys.path` d'être une chaîne nommant une archive de fichier ZIP. L'archive peut contenir une arborescence de répertoires pour prendre en charge les importations de paquets, et un chemin dans l'archive peut être donné pour importer uniquement à partir d'un sous-répertoire. Par exemple, le chemin d'accès `example.zip/lib/` importerait uniquement depuis le sous-répertoire `lib/` dans l'archive.

Tous les fichiers peuvent être présents dans l'archive ZIP, mais les importateurs ne sont invoqués que pour les fichiers `.py` et `.pyc`. L'importation ZIP de modules dynamiques (`.py`, `.so`) n'est pas permise. Notez que si une archive ne contient que des fichiers `.py`, Python n'essaiera pas de modifier l'archive en ajoutant le fichier `.pyc` correspondant, ce qui signifie que si une archive ZIP ne contient pas de fichiers `.pyc`, l'importation peut être assez lente.

Modifié dans la version 3.13 : ZIP64 is supported

Modifié dans la version 3.8 : auparavant, les archives ZIP avec un commentaire d'archive n'étaient pas prises en charge.

Voir aussi

PKZIP Application Note
Documentation sur le format de fichier ZIP par Phil Katz, créateur du format et des algorithmes utilisés.

PEP 273 — Importation de modules depuis des archives ZIP
Écrit par James C. Ahlstrom, qui a également fourni une mise en œuvre. Python 2.3 suit les spécifications de **PEP 273**, mais utilise une implémentation écrite par Just van Rossum qui utilise les crochets d'importation décrits dans **PEP 302**.

***importlib* — Implémentation du système d'importation**
Paquet qui définit les protocoles que doivent implémenter tous les objets importateurs.

Ce module définit une exception :

exception zipimport.**ZipImportError**

Exception levée par les objets *zipimporter*. C'est une sous-classe de `ImportError`, donc elle peut être aussi interceptée comme une `ImportError`.

32.1.1 Objets *zimporter*

`zipimporter` est la classe pour importer des fichiers ZIP.

class zipimport.**zipimporter**(*archivepath*)

Classe d'objets qui importent depuis les archives. Dans le constructeur, *archivepath* doit être un chemin vers un fichier ZIP, éventuellement augmenté d'un chemin à l'intérieur de l'archive. Par exemple, un *archivepath* de `foo/bar.zip/lib` cherchera les modules dans le répertoire `lib` du fichier ZIP `foo/bar.zip` (à supposer que ce répertoire existe).

`ZipImportError` est levée si *archivepath* ne pointe pas vers une archive ZIP valide.

Modifié dans la version 3.12 : Methods `find_loader()` and `find_module()`, deprecated in 3.10 are now removed. Use `find_spec()` instead.

create_module(*spec*)

Implémentation de `importlib.abc.Loader.create_module()`. Elle renvoie toujours `None`, ce qui déclenche le mécanisme standard.

Ajouté dans la version 3.10.

exec_module(*module*)

Implémentation de `importlib.abc.Loader.exec_module()`.

Ajouté dans la version 3.10.

find_spec(*fullname*, *target=None*)

Implémentation de `importlib.abc.PathEntryFinder.find_spec()`.

Ajouté dans la version 3.10.

get_code(*fullname*)

Renvoie l'objet de code pour le module spécifié. Lève `ZipImportError` si l'importation a échoué.

get_data(*pathname*)

Renvoie les données associées à *pathname*. Lève `OSError` si le fichier n'a pas été trouvé.

Modifié dans la version 3.3 : `IOError` used to be raised, it is now an alias of `OSError`.

get_filename(*fullname*)

Renvoie la valeur `__file__` qui serait définie si le module spécifié était importé. Lève `ZipImportError` si l'importation a échoué.

Ajouté dans la version 3.1.

get_source(*fullname*)

Renvoie le code source du module spécifié. Lève `ZipImportError` si le module n'a pas pu être trouvé, renvoie `None` si l'archive contient le module, mais n'en a pas la source.

is_package(*fullname*)

Renvoie `True` si le module spécifié par *fullname* est un paquet. Lève `ZipImportError` si le module n'a pas pu être trouvé.

load_module(*fullname*)

Charge et renvoie le module spécifié par *fullname*, qui doit être le nom du module entièrement qualifié (avec des points). Lève `ZipImportError` si l'importation a échoué.

Deprecated since version 3.10, will be removed in version 3.15 : utilisez plutôt `exec_module()`.

invalidate_caches()

Efface le cache interne des informations sur les fichiers à l'intérieur de l'archive ZIP.

Ajouté dans la version 3.10.

archive

Le nom de fichier de l'archive ZIP associé à l'importateur, sans sous-chemin à l'intérieur.

prefix

Le sous-chemin du fichier ZIP où les modules sont recherchés. C'est la chaîne vide pour les objets *zipimporter* qui pointent vers la racine du fichier ZIP.

On a schématiquement `archive` + `'/'` + `prefix` == *archivepath*, où *archivepath* est l'argument donné au constructeur `zipimporter`.

32.1.2 Exemples

Voici un exemple qui importe un module d'une archive ZIP — notez que le module `zipimport` n'est pas explicitement utilisé.

```
$ unzip -l example.zip
Archive:  example.zip
  Length     Date   Time    Name
 --------    ----   ----    ----
     8467  11-26-02 22:30   jwzthreading.py
 --------                   -------
     8467                   1 file
$ ./python
Python 2.3 (#1, Aug 1 2003, 19:54:32)
>>> import sys
>>> sys.path.insert(0, 'example.zip')  # Add .zip file to front of path
>>> import jwzthreading
>>> jwzthreading.__file__
'example.zip/jwzthreading.py'
```

32.2 `pkgutil` --- Package extension utility

Code source : Lib/pkgutil.py

Ce module fournit des utilitaires pour le système d'importation, en particulier pour la prise en charge des paquets.

class pkgutil.**ModuleInfo**(*module_finder*, *name*, *ispkg*)

Un *n*-uplet nommé qui contient un bref résumé des informations d'un module.

Ajouté dans la version 3.6.

pkgutil.**extend_path**(*path*, *name*)

Étend le chemin de recherche pour les modules qui composent un paquet. L'usage prévu est de placer le code suivant dans le `__init__.py` d'un paquet :

```
from pkgutil import extend_path
__path__ = extend_path(__path__, __name__)
```

For each directory on `sys.path` that has a subdirectory that matches the package name, add the subdirectory to the package's `__path__`. This is useful if one wants to distribute different parts of a single logical package as multiple directories.

It also looks for `*.pkg` files beginning where `*` matches the *name* argument. This feature is similar to `*.pth` files (see the `site` module for more information), except that it doesn't special-case lines starting with `import`. A `*.pkg` file is trusted at face value : apart from skipping blank lines and ignoring comments, all entries found in a `*.pkg` file are added to the path, regardless of whether they exist on the filesystem (this is a feature).

Si le chemin d'entrée n'est pas une liste (comme c'est le cas pour les paquets figés), il est retourné tel quel. Le chemin d'entrée n'est pas modifié ; une copie étendue en est retournée. Les éléments ne sont ajoutés à la copie qu'à la fin.

L'interpréteur estime que `sys.path` est une séquence. Les éléments de `sys.path` qui ne sont pas des chaînes se référant à des répertoires existants sont ignorés. Les éléments Unicode sur `sys.path` qui provoquent des erreurs lorsqu'ils sont utilisés comme noms de fichiers peuvent amener cette fonction à lever une exception (conformément au comportement de `os.path.isdir()`).

`pkgutil.`**`get_importer`**(*path_item*)

Récupère un *finder* pour l'élément *path_item* donné.

Le chercheur retourné est mis en cache dans `sys.path_importer_cache` s'il a été récemment créé par le chemin d'un point d'entrée.

Le cache (ou une partie de celui-ci) peut être effacé manuellement si une nouvelle analyse de `sys.path_hooks` est nécessaire.

Modifié dans la version 3.3 : Mise à jour pour être basée directement sur `importlib` au lieu de dépendre de l'émulation interne de la **PEP 302** du paquet.

`pkgutil.`**`iter_importers`**(*fullname=''*)

Génère des objets *finder* pour le nom du module donné.

Si le nom complet contient un `'.'`, les chercheurs sont pour le paquet contenant le nom complet, sinon ils sont enregistrés pour tous les chercheurs de niveau supérieur (c'est-à-dire ceux de `sys.meta_path` et de `sys.path_hooks`).

Si le module nommé se trouve dans un paquet, ce paquet est importé en tant qu'effet secondaire de l'invocation de cette fonction.

Si aucun nom de module n'est spécifié, tous les chercheurs de niveau supérieur sont générés.

Modifié dans la version 3.3 : Mise à jour pour être basée directement sur `importlib` au lieu de dépendre de l'émulation interne de la **PEP 302** du paquet.

`pkgutil.`**`iter_modules`**(*path=None*, *prefix=''*)

Fournit des `ModuleInfo` pour tous les sous-modules sur *path* ou, si *path* est `None`, pour tous les modules de niveau supérieur sur `sys.path`.

path doit être soit `None`, soit une liste de chemins pour rechercher des modules.

prefix est une chaîne de caractères à afficher au début de chaque nom de module en sortie.

> **Note**
>
> Cela fonctionne uniquement pour un *finder* qui définit une méthode `iter_modules()`. Cette interface n'est pas standard, donc le module fournit également des implémentations pour `importlib.machinery.FileFinder` et `zipimport.zipimporter`.

Modifié dans la version 3.3 : Mise à jour pour être basée directement sur `importlib` au lieu de dépendre de l'émulation interne de la **PEP 302** du paquet.

pkgutil.**walk_packages**(*path=None*, *prefix=''*, *onerror=None*)

Fournit des `ModuleInfo` pour tous les modules de manière récursive sur *path* ou, si *path* est `None`, tous les modules accessibles.

path doit être soit `None`, soit une liste de chemins pour rechercher des modules.

prefix est une chaîne de caractères à afficher au début de chaque nom de module en sortie.

Notez que cette fonction doit importer tous les *packages* (*pas* tous les modules !) sur le *path* donné, afin d'accéder à l'attribut `__path__` pour trouver les sous-modules.

onerror est une fonction qui est appelée avec un argument (le nom du paquet qui était en cours d'importation) si une exception se produit lors de la tentative d'importation d'un paquet. Si aucune fonction *onerror* n'est fournie, les `ImportError`s sont attrapées et ignorées, tandis que toutes les autres exceptions sont propagées, mettant fin à la recherche.

Exemples :

```
# list all modules python can access
walk_packages()

# list all submodules of ctypes
walk_packages(ctypes.__path__, ctypes.__name__ + '.')
```

Note

Cela fonctionne uniquement pour un *finder* qui définit une méthode `iter_modules()`. Cette interface n'est pas standard, donc le module fournit également des implémentations pour `importlib.machinery.FileFinder` et `zipimport.zipimporter`.

Modifié dans la version 3.3 : Mise à jour pour être basée directement sur `importlib` au lieu de dépendre de l'émulation interne de la **PEP 302** du paquet.

pkgutil.**get_data**(*package*, *resource*)

Obtient une ressource à partir d'un paquet.

Ceci est une surcouche pour l'API *loader* `get_data`. L'argument *package* doit être le nom d'un paquet, au format module standard (`foo.bar`). L'argument *resource* doit être sous forme d'un nom de fichier relatif, en utilisant `/` comme séparateur de chemin. Le nom du répertoire parent `..` n'est pas autorisé, pas plus qu'un nom racine (commençant par `/`).

La fonction renvoie une chaîne binaire qui est le contenu de la ressource spécifiée.

Pour les paquets situés dans le système de fichiers, qui ont déjà été importés, c'est l'équivalent approximatif de :

```
d = os.path.dirname(sys.modules[package].__file__)
data = open(os.path.join(d, resource), 'rb').read()
```

Si le paquet ne peut pas être localisé ou chargé, ou s'il utilise un *loader* qui ne prend pas en charge `get_data`, alors `None` est retourné. En particulier, le *loader* pour les *paquets espaces de noms* ne prend pas en charge `get_data`.

pkgutil.**resolve_name**(*name*)

Solutionne un nom en un objet.

Cette fonctionnalité est utilisée dans de nombreux endroits de la bibliothèque standard (voir bpo-12915) - et une fonctionnalité équivalente est également présente dans des paquets largement utilisés tels que Setuptools, Django et Pyramid.

Python s'attend à ce que *name* soit une chaîne de caractères dans l'un des formats suivants, où W est une abréviation pour un identifiant Python valide et le point représente un point littéral dans ces pseudo-regexes :

— `W(.W)*`

— `W(.W)*:(W(.W)*)?`

La première forme est destinée uniquement à assurer la compatibilité ascendante. Elle suppose qu'une partie du nom pointé est un paquet, et que le reste est un objet quelque part à l'intérieur de ce paquet, éventuellement niché à l'intérieur d'autres objets. Puisque l'endroit où le paquet s'arrête et où la hiérarchie des objets commence ne peut pas être déduit par inspection, des tentatives répétées d'importation doivent être effectuées avec cette forme.

Dans la deuxième forme, l'appelant clarifie le point de division en fournissant un seul deux-points : le nom pointé à gauche des deux-points est un package à importer, et le nom pointé à droite est la hiérarchie d'objets à l'intérieur de ce paquet. Seule une importation est nécessaire dans cette forme. Si elle se termine par un deux-points, alors un objet module est retourné.

La fonction renvoie un objet (qui pourrait être un module), ou génère l'une des exceptions suivantes :

`ValueError` – si *name* n'est pas un format reconnu.

`ImportError` – si une importation échoue lorsqu'elle n'aurait pas dû.

`AttributeError` – Si un échec s'est produit lors du parcours de la hiérarchie d'objets dans le paquet importé pour accéder à l'objet souhaité.

Ajouté dans la version 3.9.

32.3 `modulefinder` --- Find modules used by a script

Code source : Lib/modulefinder.py

Ce module fournit une classe `ModuleFinder` qui peut être utilisée pour déterminer la liste des modules importés par un script. `modulefinder.py` peut aussi être utilisé en tant que script, en passant le nom du fichier Python en argument, ce qui affichera un rapport sur les modules importés.

`modulefinder.`**`AddPackagePath`**(*pkg_name*, *path*)

Enregistre que le paquet *pkg_name* peut être trouvé au chemin *path* spécifié.

`modulefinder.`**`ReplacePackage`**(*oldname*, *newname*)

Permet de spécifier que le module nommé *oldname* est en réalité le paquet nommé *newname*.

`class` `modulefinder.`**`ModuleFinder`**(*path=None*, *debug=0*, *excludes=[]*, *replace_paths=[]*)

Cette classe fournit les méthodes `run_script()` et `report()` pour déterminer l'ensemble des modules importés par un script. *path* peut être une liste de dossiers dans lesquels chercher les modules ; si non spécifié, `sys.path` est utilisé. *debug* définit le niveau de débogage ; des valeurs plus élevées produisent plus de détails sur ce que fait la classe. *excludes* est une liste de noms de modules à exclure de l'analyse. *replace_paths* est une liste de paires `(oldpath, newpath)` qui seront remplacés dans les chemins des modules.

`report`()

Affiche un rapport sur la sortie standard qui liste les modules importés par le script et leurs chemins, ainsi que les modules manquants ou qui n'ont pas été trouvés.

`run_script`(*pathname*)

Analyse le contenu du fichier *pathname*, qui doit contenir du code Python.

`modules`

Un dictionnaire de correspondance entre nom de modules et modules. Voir *Exemples d'utilisation de la classe ModuleFinder*.

32.3.1 Exemples d'utilisation de la classe `ModuleFinder`

Le script qui sera analysé (*bacon.py*) :

```
import re, itertools

try:
    import baconhameggs
except ImportError:
    pass

try:
    import guido.python.ham
except ImportError:
    pass
```

Le script qui va afficher le rapport de *bacon.py* :

```
from modulefinder import ModuleFinder

finder = ModuleFinder()
finder.run_script('bacon.py')

print('Loaded modules:')
for name, mod in finder.modules.items():
    print('%s: ' % name, end='')
    print(',''.join(list(mod.globalnames.keys())[:3]))

print('-'*50)
print('Modules not imported:')
print('\n'.join(finder.badmodules.keys()))
```

Exemple de sortie (peut varier en fonction de l'architecture) :

```
Loaded modules:
_types:
copyreg:  _inverted_registry,_slotnames,__all__
re._compiler:  isstring,_sre,_optimize_unicode
_sre:
re._constants:  REPEAT_ONE,makedict,AT_END_LINE
sys:
re:  __module__,finditer,_expand
itertools:
__main__:  re,itertools,baconhameggs
re._parser:  _PATTERNENDERS,SRE_FLAG_UNICODE
array:
types:  __module__,IntType,TypeType
--------------------------------------------------
Modules not imported:
guido.python.ham
baconhameggs
```

32.4 `runpy` --- Locating and executing Python modules

Code source : Lib/runpy.py

The *runpy* module is used to locate and run Python modules without importing them first. Its main use is to implement the `-m` command line switch that allows scripts to be located using the Python module namespace rather than the filesystem.

Note that this is *not* a sandbox module - all code is executed in the current process, and any side effects (such as cached imports of other modules) will remain in place after the functions have returned.

Furthermore, any functions and classes defined by the executed code are not guaranteed to work correctly after a `runpy` function has returned. If that limitation is not acceptable for a given use case, `importlib` is likely to be a more suitable choice than this module.

The `runpy` module provides two functions :

`runpy.run_module`(*mod_name*, *init_globals=None*, *run_name=None*, *alter_sys=False*)

Execute the code of the specified module and return the resulting module's globals dictionary. The module's code is first located using the standard import mechanism (refer to **PEP 302** for details) and then executed in a fresh module namespace.

The *mod_name* argument should be an absolute module name. If the module name refers to a package rather than a normal module, then that package is imported and the `__main__` submodule within that package is then executed and the resulting module globals dictionary returned.

The optional dictionary argument *init_globals* may be used to pre-populate the module's globals dictionary before the code is executed. *init_globals* will not be modified. If any of the special global variables below are defined in *init_globals*, those definitions are overridden by `run_module()`.

The special global variables `__name__`, `__spec__`, `__file__`, `__cached__`, `__loader__` and `__package__` are set in the globals dictionary before the module code is executed. (Note that this is a minimal set of variables - other variables may be set implicitly as an interpreter implementation detail.)

`__name__` is set to *run_name* if this optional argument is not `None`, to `mod_name + '.__main__'` if the named module is a package and to the *mod_name* argument otherwise.

`__spec__` will be set appropriately for the *actually* imported module (that is, `__spec__.name` will always be *mod_name* or `mod_name + '.__main__'`, never *run_name*).

`__file__`, `__cached__`, `__loader__` and `__package__` are set as normal based on the module spec.

If the argument *alter_sys* is supplied and evaluates to `True`, then `sys.argv[0]` is updated with the value of `__file__` and `sys.modules[__name__]` is updated with a temporary module object for the module being executed. Both `sys.argv[0]` and `sys.modules[__name__]` are restored to their original values before the function returns.

Note that this manipulation of `sys` is not thread-safe. Other threads may see the partially initialised module, as well as the altered list of arguments. It is recommended that the `sys` module be left alone when invoking this function from threaded code.

Voir aussi

The `-m` option offering equivalent functionality from the command line.

Modifié dans la version 3.1 : Added ability to execute packages by looking for a `__main__` submodule.

Modifié dans la version 3.2 : Added `__cached__` global variable (see **PEP 3147**).

Modifié dans la version 3.4 : Updated to take advantage of the module spec feature added by **PEP 451**. This allows `__cached__` to be set correctly for modules run this way, as well as ensuring the real module name is always accessible as `__spec__.name`.

Modifié dans la version 3.12 : The setting of `__cached__`, `__loader__`, and `__package__` are deprecated. See `ModuleSpec` for alternatives.

`runpy.run_path`(*path_name*, *init_globals=None*, *run_name=None*)

Execute the code at the named filesystem location and return the resulting module's globals dictionary. As with a script name supplied to the CPython command line, *file_path* may refer to a Python source file, a compiled bytecode file or a valid `sys.path` entry containing a `__main__` module (e.g. a zipfile containing a top-level `__main__.py` file).

For a simple script, the specified code is simply executed in a fresh module namespace. For a valid `sys.path`

entry (typically a zipfile or directory), the entry is first added to the beginning of `sys.path`. The function then looks for and executes a `__main__` module using the updated path. Note that there is no special protection against invoking an existing `__main__` entry located elsewhere on `sys.path` if there is no such module at the specified location.

The optional dictionary argument *init_globals* may be used to pre-populate the module's globals dictionary before the code is executed. *init_globals* will not be modified. If any of the special global variables below are defined in *init_globals*, those definitions are overridden by `run_path()`.

The special global variables `__name__`, `__spec__`, `__file__`, `__cached__`, `__loader__` and `__package__` are set in the globals dictionary before the module code is executed. (Note that this is a minimal set of variables - other variables may be set implicitly as an interpreter implementation detail.)

`__name__` is set to *run_name* if this optional argument is not `None` and to `'<run_path>'` otherwise.

If *file_path* directly references a script file (whether as source or as precompiled byte code), then `__file__` will be set to *file_path*, and `__spec__`, `__cached__`, `__loader__` and `__package__` will all be set to `None`.

If *file_path* is a reference to a valid `sys.path` entry, then `__spec__` will be set appropriately for the imported `__main__` module (that is, `__spec__.name` will always be `__main__`). `__file__`, `__cached__`, `__loader__` and `__package__` will be set as normal based on the module spec.

A number of alterations are also made to the `sys` module. Firstly, `sys.path` may be altered as described above. `sys.argv[0]` is updated with the value of *file_path* and `sys.modules[__name__]` is updated with a temporary module object for the module being executed. All modifications to items in `sys` are reverted before the function returns.

Note that, unlike `run_module()`, the alterations made to `sys` are not optional in this function as these adjustments are essential to allowing the execution of `sys.path` entries. As the thread-safety limitations still apply, use of this function in threaded code should be either serialised with the import lock or delegated to a separate process.

Voir aussi

using-on-interface-options for equivalent functionality on the command line (`python path/to/script`).

Ajouté dans la version 3.2.

Modifié dans la version 3.4 : Updated to take advantage of the module spec feature added by **PEP 451**. This allows `__cached__` to be set correctly in the case where `__main__` is imported from a valid `sys.path` entry rather than being executed directly.

Modifié dans la version 3.12 : The setting of `__cached__`, `__loader__`, and `__package__` are deprecated.

Voir aussi

PEP 338 -- Exécuter des modules en tant que scripts
PEP written and implemented by Nick Coghlan.

PEP 366 -- Main module explicit relative imports
PEP written and implemented by Nick Coghlan.

PEP 451 -- A ModuleSpec Type for the Import System
PEP written and implemented by Eric Snow

using-on-general - CPython command line details

The `importlib.import_module()` function

32.5 importlib --- The implementation of import

Ajouté dans la version 3.1.

Source code : Lib/importlib/__init__.py

32.5.1 Introduction

The purpose of the *importlib* package is three-fold.

One is to provide the implementation of the import statement (and thus, by extension, the *__import__()* function) in Python source code. This provides an implementation of import which is portable to any Python interpreter. This also provides an implementation which is easier to comprehend than one implemented in a programming language other than Python.

Two, the components to implement import are exposed in this package, making it easier for users to create their own custom objects (known generically as an *importer*) to participate in the import process.

Three, the package contains modules exposing additional functionality for managing aspects of Python packages :

- *importlib.metadata* presents access to metadata from third-party distributions.
- *importlib.resources* provides routines for accessing non-code "resources" from Python packages.

Voir aussi

import
: The language reference for the import statement.

Packages specification
: Original specification of packages. Some semantics have changed since the writing of this document (e.g. redirecting based on None in *sys.modules*).

La fonction *__import__()*
: The import statement is syntactic sugar for this function.

The initialization of the sys.path module search path
: The initialization of *sys.path*.

PEP 235
: Import on Case-Insensitive Platforms

PEP 263
: Defining Python Source Code Encodings

PEP 302
: New Import Hooks

PEP 328
: Imports : Multi-Line and Absolute/Relative

PEP 366
: Main module explicit relative imports

PEP 420
: Implicit namespace packages

PEP 451
: A ModuleSpec Type for the Import System

PEP 488
: Elimination of PYO files

PEP 489
: Multi-phase extension module initialization

PEP 552
: Deterministic pycs

PEP 3120
: Using UTF-8 as the Default Source Encoding

PEP 3147
: PYC Repository Directories

32.5.2 Fonctions

`importlib.__import__`(*name*, *globals=None*, *locals=None*, *fromlist=()*, *level=0*)

An implementation of the built-in `__import__()` function.

> **Note**
>
> Programmatic importing of modules should use `import_module()` instead of this function.

`importlib.import_module`(*name*, *package=None*)

Import a module. The *name* argument specifies what module to import in absolute or relative terms (e.g. either `pkg.mod` or `..mod`). If the name is specified in relative terms, then the *package* argument must be set to the name of the package which is to act as the anchor for resolving the package name (e.g. `import_module('..mod', 'pkg.subpkg')` will import `pkg.mod`).

The `import_module()` function acts as a simplifying wrapper around `importlib.__import__()`. This means all semantics of the function are derived from `importlib.__import__()`. The most important difference between these two functions is that `import_module()` returns the specified package or module (e.g. `pkg.mod`), while `__import__()` returns the top-level package or module (e.g. `pkg`).

If you are dynamically importing a module that was created since the interpreter began execution (e.g., created a Python source file), you may need to call `invalidate_caches()` in order for the new module to be noticed by the import system.

Modifié dans la version 3.3 : Parent packages are automatically imported.

`importlib.invalidate_caches`()

Invalidate the internal caches of finders stored at `sys.meta_path`. If a finder implements `invalidate_caches()` then it will be called to perform the invalidation. This function should be called if any modules are created/installed while your program is running to guarantee all finders will notice the new module's existence.

Ajouté dans la version 3.3.

Modifié dans la version 3.10 : Namespace packages created/installed in a different `sys.path` location after the same namespace was already imported are noticed.

`importlib.reload`(*module*)

Reload a previously imported *module*. The argument must be a module object, so it must have been successfully imported before. This is useful if you have edited the module source file using an external editor and want to try out the new version without leaving the Python interpreter. The return value is the module object (which can be different if re-importing causes a different object to be placed in `sys.modules`).

When `reload()` is executed :

- Python module's code is recompiled and the module-level code re-executed, defining a new set of objects which are bound to names in the module's dictionary by reusing the *loader* which originally loaded the module. The `init` function of extension modules is not called a second time.
- As with all other objects in Python the old objects are only reclaimed after their reference counts drop to zero.
- The names in the module namespace are updated to point to any new or changed objects.
- Other references to the old objects (such as names external to the module) are not rebound to refer to the new objects and must be updated in each namespace where they occur if that is desired.

There are a number of other caveats :

When a module is reloaded, its dictionary (containing the module's global variables) is retained. Redefinitions of names will override the old definitions, so this is generally not a problem. If the new version of a module does not define a name that was defined by the old version, the old definition remains. This feature can be used to the module's advantage if it maintains a global table or cache of objects --- with a `try` statement it can test for the table's presence and skip its initialization if desired :

```
try:
    cache
except NameError:
    cache = {}
```

It is generally not very useful to reload built-in or dynamically loaded modules. Reloading `sys`, `__main__`, `builtins` and other key modules is not recommended. In many cases extension modules are not designed to be initialized more than once, and may fail in arbitrary ways when reloaded.

If a module imports objects from another module using `from` ... `import` ..., calling `reload()` for the other module does not redefine the objects imported from it --- one way around this is to re-execute the `from` statement, another is to use `import` and qualified names (*module.name*) instead.

If a module instantiates instances of a class, reloading the module that defines the class does not affect the method definitions of the instances --- they continue to use the old class definition. The same is true for derived classes.

Ajouté dans la version 3.4.

Modifié dans la version 3.7 : `ModuleNotFoundError` is raised when the module being reloaded lacks a `ModuleSpec`.

32.5.3 `importlib.abc` -- Abstract base classes related to import

Source code : Lib/importlib/abc.py

The `importlib.abc` module contains all of the core abstract base classes used by `import`. Some subclasses of the core abstract base classes are also provided to help in implementing the core ABCs.

ABC hierarchy :

```
object
 +-- MetaPathFinder
 +-- PathEntryFinder
 +-- Loader
      +-- ResourceLoader --------+
      +-- InspectLoader          |
           +-- ExecutionLoader --+
                                 +-- FileLoader
                                 +-- SourceLoader
```

class `importlib.abc.`**`MetaPathFinder`**

An abstract base class representing a *meta path finder*.

Ajouté dans la version 3.3.

Modifié dans la version 3.10 : No longer a subclass of `Finder`.

`find_spec`(*fullname*, *path*, *target=None*)

An abstract method for finding a *spec* for the specified module. If this is a top-level import, *path* will be `None`. Otherwise, this is a search for a subpackage or module and *path* will be the value of `__path__` from the parent package. If a spec cannot be found, `None` is returned. When passed in, `target` is a module object that the finder may use to make a more educated guess about what spec to return. `importlib.util.spec_from_loader()` may be useful for implementing concrete `MetaPathFinders`.

Ajouté dans la version 3.4.

`invalidate_caches`()

An optional method which, when called, should invalidate any internal cache used by the finder. Used by `importlib.invalidate_caches()` when invalidating the caches of all finders on `sys.meta_path`.

Modifié dans la version 3.4 : Returns `None` when called instead of `NotImplemented`.

class importlib.abc.**PathEntryFinder**

An abstract base class representing a *path entry finder*. Though it bears some similarities to `MetaPathFinder`, `PathEntryFinder` is meant for use only within the path-based import subsystem provided by `importlib.machinery.PathFinder`.

Ajouté dans la version 3.3.

Modifié dans la version 3.10 : No longer a subclass of `Finder`.

find_spec(*fullname*, *target=None*)

An abstract method for finding a *spec* for the specified module. The finder will search for the module only within the *path entry* to which it is assigned. If a spec cannot be found, `None` is returned. When passed in, `target` is a module object that the finder may use to make a more educated guess about what spec to return. `importlib.util.spec_from_loader()` may be useful for implementing concrete `PathEntryFinders`.

Ajouté dans la version 3.4.

invalidate_caches()

An optional method which, when called, should invalidate any internal cache used by the finder. Used by `importlib.machinery.PathFinder.invalidate_caches()` when invalidating the caches of all cached finders.

class importlib.abc.**Loader**

An abstract base class for a *loader*. See **PEP 302** for the exact definition for a loader.

Loaders that wish to support resource reading should implement a `get_resource_reader()` method as specified by `importlib.resources.abc.ResourceReader`.

Modifié dans la version 3.7 : Introduced the optional `get_resource_reader()` method.

create_module(*spec*)

A method that returns the module object to use when importing a module. This method may return `None`, indicating that default module creation semantics should take place.

Ajouté dans la version 3.4.

Modifié dans la version 3.6 : This method is no longer optional when `exec_module()` is defined.

exec_module(*module*)

An abstract method that executes the module in its own namespace when a module is imported or reloaded. The module should already be initialized when `exec_module()` is called. When this method exists, `create_module()` must be defined.

Ajouté dans la version 3.4.

Modifié dans la version 3.6 : `create_module()` must also be defined.

load_module(*fullname*)

A legacy method for loading a module. If the module cannot be loaded, `ImportError` is raised, otherwise the loaded module is returned.

If the requested module already exists in `sys.modules`, that module should be used and reloaded. Otherwise the loader should create a new module and insert it into `sys.modules` before any loading begins, to prevent recursion from the import. If the loader inserted a module and the load fails, it must be removed by the loader from `sys.modules`; modules already in `sys.modules` before the loader began execution should be left alone.

The loader should set several attributes on the module (note that some of these attributes can change when a module is reloaded) :
— `module.__name__`
— `module.__file__`
— `module.__cached__` *(deprecated)*
— `module.__path__`
— `module.__package__` *(deprecated)*

— `module.__loader__` *(deprecated)*
When `exec_module()` is available then backwards-compatible functionality is provided.

Modifié dans la version 3.4 : Raise `ImportError` when called instead of `NotImplementedError`. Functionality provided when `exec_module()` is available.

Deprecated since version 3.4, will be removed in version 3.15 : The recommended API for loading a module is `exec_module()` (and `create_module()`). Loaders should implement it instead of `load_module()`. The import machinery takes care of all the other responsibilities of `load_module()` when `exec_module()` is implemented.

class `importlib.abc.`**`ResourceLoader`**

Superseded by TraversableResources

An abstract base class for a *loader* which implements the optional **PEP 302** protocol for loading arbitrary resources from the storage back-end.

Obsolète depuis la version 3.7 : This ABC is deprecated in favour of supporting resource loading through `importlib.resources.abc.TraversableResources`.

abstractmethod get_data(*path*)

An abstract method to return the bytes for the data located at *path*. Loaders that have a file-like storage back-end that allows storing arbitrary data can implement this abstract method to give direct access to the data stored. `OSError` is to be raised if the *path* cannot be found. The *path* is expected to be constructed using a module's `__file__` attribute or an item from a package's `__path__`.

Modifié dans la version 3.4 : Raises `OSError` instead of `NotImplementedError`.

class `importlib.abc.`**`InspectLoader`**

An abstract base class for a *loader* which implements the optional **PEP 302** protocol for loaders that inspect modules.

get_code(*fullname*)

Return the code object for a module, or `None` if the module does not have a code object (as would be the case, for example, for a built-in module). Raise an `ImportError` if loader cannot find the requested module.

> **Note**
>
> While the method has a default implementation, it is suggested that it be overridden if possible for performance.

Modifié dans la version 3.4 : No longer abstract and a concrete implementation is provided.

abstractmethod get_source(*fullname*)

An abstract method to return the source of a module. It is returned as a text string using *universal newlines*, translating all recognized line separators into `'\n'` characters. Returns `None` if no source is available (e.g. a built-in module). Raises `ImportError` if the loader cannot find the module specified.

Modifié dans la version 3.4 : Raises `ImportError` instead of `NotImplementedError`.

is_package(*fullname*)

An optional method to return a true value if the module is a package, a false value otherwise. `ImportError` is raised if the *loader* cannot find the module.

Modifié dans la version 3.4 : Raises `ImportError` instead of `NotImplementedError`.

static source_to_code(*data*, *path='<string>'*)

Create a code object from Python source.

The *data* argument can be whatever the `compile()` function supports (i.e. string or bytes). The *path* argument should be the "path" to where the source code originated from, which can be an abstract concept (e.g. location in a zip file).

With the subsequent code object one can execute it in a module by running `exec(code, module.__dict__)`.

Ajouté dans la version 3.4.

Modifié dans la version 3.5 : Made the method static.

exec_module(*module*)

Implementation of `Loader.exec_module()`.

Ajouté dans la version 3.4.

load_module(*fullname*)

Implementation of `Loader.load_module()`.

Deprecated since version 3.4, will be removed in version 3.15 : use `exec_module()` instead.

class importlib.abc.**ExecutionLoader**

An abstract base class which inherits from `InspectLoader` that, when implemented, helps a module to be executed as a script. The ABC represents an optional **PEP 302** protocol.

abstractmethod get_filename(*fullname*)

An abstract method that is to return the value of `__file__` for the specified module. If no path is available, `ImportError` is raised.

If source code is available, then the method should return the path to the source file, regardless of whether a bytecode was used to load the module.

Modifié dans la version 3.4 : Raises `ImportError` instead of `NotImplementedError`.

class importlib.abc.**FileLoader**(*fullname*, *path*)

An abstract base class which inherits from `ResourceLoader` and `ExecutionLoader`, providing concrete implementations of `ResourceLoader.get_data()` and `ExecutionLoader.get_filename()`.

The *fullname* argument is a fully resolved name of the module the loader is to handle. The *path* argument is the path to the file for the module.

Ajouté dans la version 3.3.

name

The name of the module the loader can handle.

path

Path to the file of the module.

load_module(*fullname*)

Calls super's `load_module()`.

Deprecated since version 3.4, will be removed in version 3.15 : Use `Loader.exec_module()` instead.

abstractmethod get_filename(*fullname*)

Returns `path`.

abstractmethod get_data(*path*)

Reads *path* as a binary file and returns the bytes from it.

class importlib.abc.**SourceLoader**

An abstract base class for implementing source (and optionally bytecode) file loading. The class inherits from both `ResourceLoader` and `ExecutionLoader`, requiring the implementation of :

— `ResourceLoader.get_data()`

— ***ExecutionLoader.get_filename()***

Should only return the path to the source file ; sourceless loading is not supported.

The abstract methods defined by this class are to add optional bytecode file support. Not implementing these optional methods (or causing them to raise `NotImplementedError`) causes the loader to only work with source code. Implementing the methods allows the loader to work with source *and* bytecode files ; it does not

allow for *sourceless* loading where only bytecode is provided. Bytecode files are an optimization to speed up loading by removing the parsing step of Python's compiler, and so no bytecode-specific API is exposed.

path_stats(*path*)

Optional abstract method which returns a `dict` containing metadata about the specified path. Supported dictionary keys are :

— `'mtime'` (mandatory) : an integer or floating-point number representing the modification time of the source code ;

— `'size'` (optional) : the size in bytes of the source code.

Any other keys in the dictionary are ignored, to allow for future extensions. If the path cannot be handled, `OSError` is raised.

Ajouté dans la version 3.3.

Modifié dans la version 3.4 : Raise `OSError` instead of `NotImplementedError`.

path_mtime(*path*)

Optional abstract method which returns the modification time for the specified path.

Obsolète depuis la version 3.3 : This method is deprecated in favour of `path_stats()`. You don't have to implement it, but it is still available for compatibility purposes. Raise `OSError` if the path cannot be handled.

Modifié dans la version 3.4 : Raise `OSError` instead of `NotImplementedError`.

set_data(*path*, *data*)

Optional abstract method which writes the specified bytes to a file path. Any intermediate directories which do not exist are to be created automatically.

When writing to the path fails because the path is read-only (`errno.EACCES`/`PermissionError`), do not propagate the exception.

Modifié dans la version 3.4 : No longer raises `NotImplementedError` when called.

get_code(*fullname*)

Concrete implementation of `InspectLoader.get_code()`.

exec_module(*module*)

Concrete implementation of `Loader.exec_module()`.

Ajouté dans la version 3.4.

load_module(*fullname*)

Concrete implementation of `Loader.load_module()`.

Deprecated since version 3.4, will be removed in version 3.15 : utilisez plutôt `exec_module()`.

get_source(*fullname*)

Concrete implementation of `InspectLoader.get_source()`.

is_package(*fullname*)

Concrete implementation of `InspectLoader.is_package()`. A module is determined to be a package if its file path (as provided by `ExecutionLoader.get_filename()`) is a file named `__init__` when the file extension is removed **and** the module name itself does not end in `__init__`.

class importlib.abc.**ResourceReader**

Superseded by TraversableResources

An *abstract base class* to provide the ability to read *resources*.

From the perspective of this ABC, a *resource* is a binary artifact that is shipped within a package. Typically this is something like a data file that lives next to the `__init__.py` file of the package. The purpose of this class is to help abstract out the accessing of such data files so that it does not matter if the package and its data file(s) are stored e.g. in a zip file versus on the file system.

For any of methods of this class, a *resource* argument is expected to be a *path-like object* which represents

conceptually just a file name. This means that no subdirectory paths should be included in the *resource* argument. This is because the location of the package the reader is for, acts as the "directory". Hence the metaphor for directories and file names is packages and resources, respectively. This is also why instances of this class are expected to directly correlate to a specific package (instead of potentially representing multiple packages or a module).

Loaders that wish to support resource reading are expected to provide a method called `get_resource_reader(fullname)` which returns an object implementing this ABC's interface. If the module specified by fullname is not a package, this method should return `None`. An object compatible with this ABC should only be returned when the specified module is a package.

Ajouté dans la version 3.7.

Deprecated since version 3.12, removed in version 3.14 : Use `importlib.resources.abc.TraversableResources` instead.

abstractmethod open_resource(*resource*)

Returns an opened, *file-like object* for binary reading of the *resource*.

Si la ressource ne peut pas être trouvée, `FileNotFoundError` est levée.

abstractmethod resource_path(*resource*)

Renvoie le chemin de *resource* dans le système de fichiers.

If the resource does not concretely exist on the file system, raise `FileNotFoundError`.

abstractmethod is_resource(*name*)

Returns `True` if the named *name* is considered a resource. `FileNotFoundError` is raised if *name* does not exist.

abstractmethod contents()

Returns an *iterable* of strings over the contents of the package. Do note that it is not required that all names returned by the iterator be actual resources, e.g. it is acceptable to return names for which `is_resource()` would be false.

Allowing non-resource names to be returned is to allow for situations where how a package and its resources are stored are known a priori and the non-resource names would be useful. For instance, returning subdirectory names is allowed so that when it is known that the package and resources are stored on the file system then those subdirectory names can be used directly.

The abstract method returns an iterable of no items.

class importlib.abc.**Traversable**

An object with a subset of `pathlib.Path` methods suitable for traversing directories and opening files.

For a representation of the object on the file-system, use `importlib.resources.as_file()`.

Ajouté dans la version 3.9.

Deprecated since version 3.12, removed in version 3.14 : Use `importlib.resources.abc.Traversable` instead.

name

Abstract. The base name of this object without any parent references.

abstractmethod iterdir()

Yield `Traversable` objects in `self`.

abstractmethod is_dir()

Return `True` if `self` is a directory.

abstractmethod is_file()

Return `True` if `self` is a file.

abstractmethod joinpath(*child*)

Return Traversable child in `self`.

abstractmethod __truediv__(*child*)

Return `Traversable` child in `self`.

abstractmethod open(*mode='r'*, **args*, ***kwargs*)

mode may be 'r' or 'rb' to open as text or binary. Return a handle suitable for reading (same as `pathlib.Path.open`).

When opening as text, accepts encoding parameters such as those accepted by `io.TextIOWrapper`.

read_bytes()

Read contents of `self` as bytes.

read_text(*encoding=None*)

Read contents of `self` as text.

class importlib.abc.**TraversableResources**

An abstract base class for resource readers capable of serving the `importlib.resources.files()` interface. Subclasses `importlib.resources.abc.ResourceReader` and provides concrete implementations of the `importlib.resources.abc.ResourceReader`'s abstract methods. Therefore, any loader supplying `importlib.abc.TraversableResources` also supplies ResourceReader.

Loaders that wish to support resource reading are expected to implement this interface.

Ajouté dans la version 3.9.

Deprecated since version 3.12, removed in version 3.14 : Use `importlib.resources.abc.TraversableResources` instead.

abstractmethod files()

Returns a `importlib.resources.abc.Traversable` object for the loaded package.

32.5.4 `importlib.machinery` -- Importers and path hooks

Source code : Lib/importlib/machinery.py

This module contains the various objects that help `import` find and load modules.

importlib.machinery.**SOURCE_SUFFIXES**

A list of strings representing the recognized file suffixes for source modules.

Ajouté dans la version 3.3.

importlib.machinery.**DEBUG_BYTECODE_SUFFIXES**

A list of strings representing the file suffixes for non-optimized bytecode modules.

Ajouté dans la version 3.3.

Obsolète depuis la version 3.5 : Use `BYTECODE_SUFFIXES` instead.

importlib.machinery.**OPTIMIZED_BYTECODE_SUFFIXES**

A list of strings representing the file suffixes for optimized bytecode modules.

Ajouté dans la version 3.3.

Obsolète depuis la version 3.5 : Use `BYTECODE_SUFFIXES` instead.

importlib.machinery.**BYTECODE_SUFFIXES**

A list of strings representing the recognized file suffixes for bytecode modules (including the leading dot).

Ajouté dans la version 3.3.

Modifié dans la version 3.5 : The value is no longer dependent on `__debug__`.

importlib.machinery.**EXTENSION_SUFFIXES**

A list of strings representing the recognized file suffixes for extension modules.

Ajouté dans la version 3.3.

importlib.machinery.**all_suffixes**()

Returns a combined list of strings representing all file suffixes for modules recognized by the standard import machinery. This is a helper for code which simply needs to know if a filesystem path potentially refers to a module without needing any details on the kind of module (for example, `inspect.getmodulename()`).

Ajouté dans la version 3.3.

class importlib.machinery.**BuiltinImporter**

An *importer* for built-in modules. All known built-in modules are listed in `sys.builtin_module_names`. This class implements the `importlib.abc.MetaPathFinder` and `importlib.abc.InspectLoader` ABCs.

Only class methods are defined by this class to alleviate the need for instantiation.

Modifié dans la version 3.5 : As part of **PEP 489**, the builtin importer now implements `Loader.create_module()` and `Loader.exec_module()`

class importlib.machinery.**FrozenImporter**

An *importer* for frozen modules. This class implements the `importlib.abc.MetaPathFinder` and `importlib.abc.InspectLoader` ABCs.

Only class methods are defined by this class to alleviate the need for instantiation.

Modifié dans la version 3.4 : Gained `create_module()` and `exec_module()` methods.

class importlib.machinery.**WindowsRegistryFinder**

Finder for modules declared in the Windows registry. This class implements the `importlib.abc.MetaPathFinder` ABC.

Only class methods are defined by this class to alleviate the need for instantiation.

Ajouté dans la version 3.3.

Obsolète depuis la version 3.6 : Use `site` configuration instead. Future versions of Python may not enable this finder by default.

class importlib.machinery.**PathFinder**

A *Finder* for `sys.path` and package `__path__` attributes. This class implements the `importlib.abc.MetaPathFinder` ABC.

Only class methods are defined by this class to alleviate the need for instantiation.

classmethod find_spec(*fullname*, *path=None*, *target=None*)

Class method that attempts to find a *spec* for the module specified by *fullname* on `sys.path` or, if defined, on *path*. For each path entry that is searched, `sys.path_importer_cache` is checked. If a non-false object is found then it is used as the *path entry finder* to look for the module being searched for. If no entry is found in `sys.path_importer_cache`, then `sys.path_hooks` is searched for a finder for the path entry and, if found, is stored in `sys.path_importer_cache` along with being queried about the module. If no finder is ever found then `None` is both stored in the cache and returned.

Ajouté dans la version 3.4.

Modifié dans la version 3.5 : If the current working directory -- represented by an empty string -- is no longer valid then `None` is returned but no value is cached in `sys.path_importer_cache`.

classmethod invalidate_caches()

Calls `importlib.abc.PathEntryFinder.invalidate_caches()` on all finders stored in `sys.path_importer_cache` that define the method. Otherwise entries in `sys.path_importer_cache` set to `None` are deleted.

Modifié dans la version 3.7 : Entries of `None` in `sys.path_importer_cache` are deleted.

Modifié dans la version 3.4 : Calls objects in `sys.path_hooks` with the current working directory for `''` (i.e. the empty string).

class `importlib.machinery.`**`FileFinder`**(*path*, **loader_details*)

A concrete implementation of `importlib.abc.PathEntryFinder` which caches results from the file system.

The *path* argument is the directory for which the finder is in charge of searching.

The *loader_details* argument is a variable number of 2-item tuples each containing a loader and a sequence of file suffixes the loader recognizes. The loaders are expected to be callables which accept two arguments of the module's name and the path to the file found.

The finder will cache the directory contents as necessary, making stat calls for each module search to verify the cache is not outdated. Because cache staleness relies upon the granularity of the operating system's state information of the file system, there is a potential race condition of searching for a module, creating a new file, and then searching for the module the new file represents. If the operations happen fast enough to fit within the granularity of stat calls, then the module search will fail. To prevent this from happening, when you create a module dynamically, make sure to call `importlib.invalidate_caches()`.

Ajouté dans la version 3.3.

path

The path the finder will search in.

find_spec(*fullname*, *target=None*)

Attempt to find the spec to handle *fullname* within `path`.

Ajouté dans la version 3.4.

invalidate_caches()

Clear out the internal cache.

classmethod path_hook(**loader_details*)

A class method which returns a closure for use on `sys.path_hooks`. An instance of `FileFinder` is returned by the closure using the path argument given to the closure directly and *loader_details* indirectly.

If the argument to the closure is not an existing directory, `ImportError` is raised.

class `importlib.machinery.`**`SourceFileLoader`**(*fullname*, *path*)

A concrete implementation of `importlib.abc.SourceLoader` by subclassing `importlib.abc.FileLoader` and providing some concrete implementations of other methods.

Ajouté dans la version 3.3.

name

The name of the module that this loader will handle.

path

The path to the source file.

is_package(*fullname*)

Return `True` if `path` appears to be for a package.

path_stats(*path*)

Concrete implementation of `importlib.abc.SourceLoader.path_stats()`.

set_data(*path*, *data*)

Concrete implementation of `importlib.abc.SourceLoader.set_data()`.

load_module(*name=None*)

Concrete implementation of `importlib.abc.Loader.load_module()` where specifying the name of the module to load is optional.

Deprecated since version 3.6, will be removed in version 3.15 : Use `importlib.abc.Loader.exec_module()` instead.

class `importlib.machinery.`**`SourcelessFileLoader`**(*fullname*, *path*)

A concrete implementation of `importlib.abc.FileLoader` which can import bytecode files (i.e. no source code files exist).

Please note that direct use of bytecode files (and thus not source code files) inhibits your modules from being usable by all Python implementations or new versions of Python which change the bytecode format.

Ajouté dans la version 3.3.

name

The name of the module the loader will handle.

path

The path to the bytecode file.

is_package(*fullname*)

Determines if the module is a package based on `path`.

get_code(*fullname*)

Returns the code object for `name` created from `path`.

get_source(*fullname*)

Returns `None` as bytecode files have no source when this loader is used.

load_module(*name=None*)

Concrete implementation of `importlib.abc.Loader.load_module()` where specifying the name of the module to load is optional.

Deprecated since version 3.6, will be removed in version 3.15 : Use `importlib.abc.Loader.exec_module()` instead.

class `importlib.machinery.`**`ExtensionFileLoader`**(*fullname*, *path*)

A concrete implementation of `importlib.abc.ExecutionLoader` for extension modules.

The *fullname* argument specifies the name of the module the loader is to support. The *path* argument is the path to the extension module's file.

Note that, by default, importing an extension module will fail in subinterpreters if it doesn't implement multi-phase init (see **PEP 489**), even if it would otherwise import successfully.

Ajouté dans la version 3.3.

Modifié dans la version 3.12 : Multi-phase init is now required for use in subinterpreters.

name

Name of the module the loader supports.

path

Path to the extension module.

create_module(*spec*)

Creates the module object from the given specification in accordance with **PEP 489**.

Ajouté dans la version 3.5.

exec_module(*module*)

Initializes the given module object in accordance with **PEP 489**.

Ajouté dans la version 3.5.

is_package(*fullname*)

Returns `True` if the file path points to a package's `__init__` module based on `EXTENSION_SUFFIXES`.

get_code(*fullname*)

Returns `None` as extension modules lack a code object.

get_source(*fullname*)

Returns `None` as extension modules do not have source code.

get_filename(*fullname*)

Returns *path*.

Ajouté dans la version 3.4.

class `importlib.machinery.`**NamespaceLoader**(*name*, *path*, *path_finder*)

A concrete implementation of *importlib.abc.InspectLoader* for namespace packages. This is an alias for a private class and is only made public for introspecting the `__loader__` attribute on namespace packages :

```
>>> from importlib.machinery import NamespaceLoader
>>> import my_namespace
>>> isinstance(my_namespace.__loader__, NamespaceLoader)
True
>>> import importlib.abc
>>> isinstance(my_namespace.__loader__, importlib.abc.Loader)
True
```

Ajouté dans la version 3.11.

class `importlib.machinery.`**ModuleSpec**(*name*, *loader*, **, origin=None*, *loader_state=None*, *is_package=None*)

A specification for a module's import-system-related state. This is typically exposed as the module's `__spec__` attribute. Many of these attributes are also available directly on a module : for example, `module.__spec__.origin == module.__file__`. Note, however, that while the *values* are usually equivalent, they can differ since there is no synchronization between the two objects. For example, it is possible to update the module's `__file__` at runtime and this will not be automatically reflected in the module's *__spec__.origin*, and vice versa.

Ajouté dans la version 3.4.

name

The module's fully qualified name (see `module.__name__`). The *finder* should always set this attribute to a non-empty string.

loader

The *loader* used to load the module (see `module.__loader__`). The *finder* should always set this attribute.

origin

The location the *loader* should use to load the module (see `module.__file__`). For example, for modules loaded from a `.py` file this is the filename. The *finder* should always set this attribute to a meaningful value for the *loader* to use. In the uncommon case that there is not one (like for namespace packages), it should be set to `None`.

submodule_search_locations

A (possibly empty) *sequence* of strings enumerating the locations in which a package's submodules will be found (see `module.__path__`). Most of the time there will only be a single directory in this list.

The *finder* should set this attribute to a sequence, even an empty one, to indicate to the import system that the module is a package. It should be set to `None` for non-package modules. It is set automatically later to a special object for namespace packages.

loader_state

The *finder* may set this attribute to an object containing additional, module-specific data to use when loading the module. Otherwise it should be set to `None`.

cached

The filename of a compiled version of the module's code (see `module.__cached__`). The *finder* should always set this attribute but it may be `None` for modules that do not need compiled code stored.

parent

(Read-only) The fully qualified name of the package the module is in (or the empty string for a top-level module). See `module.__package__`. If the module is a package then this is the same as *name*.

has_location

`True` if the spec's *origin* refers to a loadable location, `False` otherwise. This value impacts how `origin` is interpreted and how the module's `__file__` is populated.

class `importlib.machinery.`**`AppleFrameworkLoader`**(*name*, *path*)

A specialization of *importlib.machinery.ExtensionFileLoader* that is able to load extension modules in Framework format.

For compatibility with the iOS App Store, *all* binary modules in an iOS app must be dynamic libraries, contained in a framework with appropriate metadata, stored in the `Frameworks` folder of the packaged app. There can be only a single binary per framework, and there can be no executable binary material outside the Frameworks folder.

To accommodate this requirement, when running on iOS, extension module binaries are *not* packaged as `.so` files on `sys.path`, but as individual standalone frameworks. To discover those frameworks, this loader is be registered against the `.fwork` file extension, with a `.fwork` file acting as a placeholder in the original location of the binary on `sys.path`. The `.fwork` file contains the path of the actual binary in the `Frameworks` folder, relative to the app bundle. To allow for resolving a framework-packaged binary back to the original location, the framework is expected to contain a `.origin` file that contains the location of the `.fwork` file, relative to the app bundle.

For example, consider the case of an import `from foo.bar import _whiz`, where `_whiz` is implemented with the binary module `sources/foo/bar/_whiz.abi3.so`, with `sources` being the location registered on `sys.path`, relative to the application bundle. This module *must* be distributed as `Frameworks/foo.bar._whiz.framework/foo.bar._whiz` (creating the framework name from the full import path of the module), with an `Info.plist` file in the `.framework` directory identifying the binary as a framework. The `foo.bar._whiz` module would be represented in the original location with a `sources/foo/bar/_whiz.abi3.fwork` marker file, containing the path `Frameworks/foo.bar._whiz/foo.bar._whiz`. The framework would also contain `Frameworks/foo.bar._whiz.framework/foo.bar._whiz.origin`, containing the path to the `.fwork` file.

When a module is loaded with this loader, the `__file__` for the module will report as the location of the `.fwork` file. This allows code to use the `__file__` of a module as an anchor for file system traversal. However, the spec origin will reference the location of the *actual* binary in the `.framework` folder.

The Xcode project building the app is responsible for converting any `.so` files from wherever they exist in the `PYTHONPATH` into frameworks in the `Frameworks` folder (including stripping extensions from the module file, the addition of framework metadata, and signing the resulting framework), and creating the `.fwork` and `.origin` files. This will usually be done with a build step in the Xcode project ; see the iOS documentation for details on how to construct this build step.

Ajouté dans la version 3.13.

Availability : iOS.

name

Name of the module the loader supports.

path

Path to the `.fwork` file for the extension module.

32.5.5 `importlib.util` -- Utility code for importers

Source code : Lib/importlib/util.py

This module contains the various objects that help in the construction of an *importer*.

importlib.util.**MAGIC_NUMBER**

The bytes which represent the bytecode version number. If you need help with loading/writing bytecode then consider *importlib.abc.SourceLoader*.

Ajouté dans la version 3.4.

importlib.util.**cache_from_source**(*path*, *debug_override=None*, *, *optimization=None*)

Return the **PEP 3147/PEP 488** path to the byte-compiled file associated with the source *path*. For example, if *path* is `/foo/bar/baz.py` the return value would be `/foo/bar/__pycache__/baz.cpython-32.pyc` for Python 3.2. The `cpython-32` string comes from the current magic tag (see `get_tag()` ; if `sys.implementation.cache_tag` is not defined then *NotImplementedError* will be raised).

The *optimization* parameter is used to specify the optimization level of the bytecode file. An empty string represents no optimization, so `/foo/bar/baz.py` with an *optimization* of `''` will result in a bytecode path of `/foo/bar/__pycache__/baz.cpython-32.pyc`. `None` causes the interpreter's optimization level to be used. Any other value's string representation is used, so `/foo/bar/baz.py` with an *optimization* of `2` will lead to the bytecode path of `/foo/bar/__pycache__/baz.cpython-32.opt-2.pyc`. The string representation of *optimization* can only be alphanumeric, else *ValueError* is raised.

The *debug_override* parameter is deprecated and can be used to override the system's value for `__debug__`. A `True` value is the equivalent of setting *optimization* to the empty string. A `False` value is the same as setting *optimization* to `1`. If both *debug_override* an *optimization* are not `None` then *TypeError* is raised.

Ajouté dans la version 3.4.

Modifié dans la version 3.5 : The *optimization* parameter was added and the *debug_override* parameter was deprecated.

Modifié dans la version 3.6 : Accepte un *path-like object*.

importlib.util.**source_from_cache**(*path*)

Given the *path* to a **PEP 3147** file name, return the associated source code file path. For example, if *path* is `/foo/bar/__pycache__/baz.cpython-32.pyc` the returned path would be `/foo/bar/baz.py`. *path* need not exist, however if it does not conform to **PEP 3147** or **PEP 488** format, a *ValueError* is raised. If `sys.implementation.cache_tag` is not defined, *NotImplementedError* is raised.

Ajouté dans la version 3.4.

Modifié dans la version 3.6 : Accepte un *path-like object*.

importlib.util.**decode_source**(*source_bytes*)

Decode the given bytes representing source code and return it as a string with universal newlines (as required by *importlib.abc.InspectLoader.get_source()*).

Ajouté dans la version 3.4.

importlib.util.**resolve_name**(*name*, *package*)

Resolve a relative module name to an absolute one.

If **name** has no leading dots, then **name** is simply returned. This allows for usage such as `importlib.util.resolve_name('sys', __spec__.parent)` without doing a check to see if the **package** argument is needed.

ImportError is raised if **name** is a relative module name but **package** is a false value (e.g. `None` or the empty string). *ImportError* is also raised if a relative name would escape its containing package (e.g. requesting `..bacon` from within the `spam` package).

Ajouté dans la version 3.3.

Modifié dans la version 3.9 : To improve consistency with import statements, raise *ImportError* instead of *ValueError* for invalid relative import attempts.

importlib.util.**find_spec**(*name*, *package=None*)

Find the *spec* for a module, optionally relative to the specified **package** name. If the module is in *sys.modules*, then `sys.modules[name].__spec__` is returned (unless the spec would be `None` or is not set,

in which case `ValueError` is raised). Otherwise a search using `sys.meta_path` is done. `None` is returned if no spec is found.

If **name** is for a submodule (contains a dot), the parent module is automatically imported.

name and **package** work the same as for `import_module()`.

Ajouté dans la version 3.4.

Modifié dans la version 3.7 : Raises `ModuleNotFoundError` instead of `AttributeError` if **package** is in fact not a package (i.e. lacks a `__path__` attribute).

`importlib.util.`**`module_from_spec`**(*spec*)

Create a new module based on **spec** and `spec.loader.create_module`.

If `spec.loader.create_module` does not return `None`, then any pre-existing attributes will not be reset. Also, no `AttributeError` will be raised if triggered while accessing **spec** or setting an attribute on the module.

This function is preferred over using `types.ModuleType` to create a new module as **spec** is used to set as many import-controlled attributes on the module as possible.

Ajouté dans la version 3.5.

`importlib.util.`**`spec_from_loader`**(*name*, *loader*, **, *origin=None*, *is_package=None*)

A factory function for creating a `ModuleSpec` instance based on a loader. The parameters have the same meaning as they do for ModuleSpec. The function uses available *loader* APIs, such as `InspectLoader.is_package()`, to fill in any missing information on the spec.

Ajouté dans la version 3.4.

`importlib.util.`**`spec_from_file_location`**(*name*, *location*, **, *loader=None*, *submodule_search_locations=None*)

A factory function for creating a `ModuleSpec` instance based on the path to a file. Missing information will be filled in on the spec by making use of loader APIs and by the implication that the module will be file-based.

Ajouté dans la version 3.4.

Modifié dans la version 3.6 : Accepte un *path-like object*.

`importlib.util.`**`source_hash`**(*source_bytes*)

Return the hash of *source_bytes* as bytes. A hash-based `.pyc` file embeds the `source_hash()` of the corresponding source file's contents in its header.

Ajouté dans la version 3.7.

`importlib.util.`**`_incompatible_extension_module_restrictions`**(**, *disable_check*)

A context manager that can temporarily skip the compatibility check for extension modules. By default the check is enabled and will fail when a single-phase init module is imported in a subinterpreter. It will also fail for a multi-phase init module that doesn't explicitly support a per-interpreter GIL, when imported in an interpreter with its own GIL.

Note that this function is meant to accommodate an unusual case ; one which is likely to eventually go away. There's is a pretty good chance this is not what you were looking for.

You can get the same effect as this function by implementing the basic interface of multi-phase init (**PEP 489**) and lying about support for multiple interpreters (or per-interpreter GIL).

> **Avertissement**
>
> Using this function to disable the check can lead to unexpected behavior and even crashes. It should only be used during extension module development.

Ajouté dans la version 3.12.

class importlib.util.**LazyLoader**(*loader*)

A class which postpones the execution of the loader of a module until the module has an attribute accessed.

This class **only** works with loaders that define `exec_module()` as control over what module type is used for the module is required. For those same reasons, the loader's `create_module()` method must return `None` or a type for which its `__class__` attribute can be mutated along with not using *slots*. Finally, modules which substitute the object placed into `sys.modules` will not work as there is no way to properly replace the module references throughout the interpreter safely ; `ValueError` is raised if such a substitution is detected.

> **Note**
>
> For projects where startup time is critical, this class allows for potentially minimizing the cost of loading a module if it is never used. For projects where startup time is not essential then use of this class is **heavily** discouraged due to error messages created during loading being postponed and thus occurring out of context.

Ajouté dans la version 3.5.

Modifié dans la version 3.6 : Began calling `create_module()`, removing the compatibility warning for `importlib.machinery.BuiltinImporter` and `importlib.machinery.ExtensionFileLoader`.

classmethod factory(*loader*)

A class method which returns a callable that creates a lazy loader. This is meant to be used in situations where the loader is passed by class instead of by instance.

```
suffixes = importlib.machinery.SOURCE_SUFFIXES
loader = importlib.machinery.SourceFileLoader
lazy_loader = importlib.util.LazyLoader.factory(loader)
finder = importlib.machinery.FileFinder(path, (lazy_loader, suffixes))
```

32.5.6 Exemples

Importing programmatically

To programmatically import a module, use `importlib.import_module()`.

```
import importlib

itertools = importlib.import_module('itertools')
```

Checking if a module can be imported

If you need to find out if a module can be imported without actually doing the import, then you should use `importlib.util.find_spec()`.

Note that if `name` is a submodule (contains a dot), `importlib.util.find_spec()` will import the parent module.

```
import importlib.util
import sys

# For illustrative purposes.
name = 'itertools'

if name in sys.modules:
    print(f"{name!r} already in sys.modules")
elif (spec := importlib.util.find_spec(name)) is not None:
    # If you chose to perform the actual import ...
    module = importlib.util.module_from_spec(spec)
```

(suite sur la page suivante)

(suite de la page précédente)

```
    sys.modules[name] = module
    spec.loader.exec_module(module)
    print(f"{name!r} has been imported")
else:
    print(f"can't find the {name!r} module")
```

Importing a source file directly

This recipe should be used with caution : it is an approximation of an import statement where the file path is specified directly, rather than `sys.path` being searched. Alternatives should first be considered first, such as modifying `sys.path` when a proper module is required, or using `runpy.run_path()` when the global namespace resulting from running a Python file is appropriate.

To import a Python source file directly from a path, use the following recipe :

```
import importlib.util
import sys


def import_from_path(module_name, file_path):
    spec = importlib.util.spec_from_file_location(module_name, file_path)
    module = importlib.util.module_from_spec(spec)
    sys.modules[module_name] = module
    spec.loader.exec_module(module)
    return module


# For illustrative purposes only (use of `json` is arbitrary).
import json
file_path = json.__file__
module_name = json.__name__

# Similar outcome as `import json`.
json = import_from_path(module_name, file_path)
```

Implementing lazy imports

The example below shows how to implement lazy imports :

```
>>> import importlib.util
>>> import sys
>>> def lazy_import(name):
...     spec = importlib.util.find_spec(name)
...     loader = importlib.util.LazyLoader(spec.loader)
...     spec.loader = loader
...     module = importlib.util.module_from_spec(spec)
...     sys.modules[name] = module
...     loader.exec_module(module)
...     return module
...
>>> lazy_typing = lazy_import("typing")
>>> #lazy_typing is a real module object,
>>> #but it is not loaded in memory yet.
>>> lazy_typing.TYPE_CHECKING
False
```

Setting up an importer

For deep customizations of import, you typically want to implement an *importer*. This means managing both the *finder* and *loader* side of things. For finders there are two flavours to choose from depending on your needs : a *meta path finder* or a *path entry finder*. The former is what you would put on `sys.meta_path` while the latter is what you create using a *path entry hook* on `sys.path_hooks` which works with `sys.path` entries to potentially create a finder. This example will show you how to register your own importers so that import will use them (for creating an importer for yourself, read the documentation for the appropriate classes defined within this package) :

```
import importlib.machinery
import sys

# For illustrative purposes only.
SpamMetaPathFinder = importlib.machinery.PathFinder
SpamPathEntryFinder = importlib.machinery.FileFinder
loader_details = (importlib.machinery.SourceFileLoader,
                  importlib.machinery.SOURCE_SUFFIXES)

# Setting up a meta path finder.
# Make sure to put the finder in the proper location in the list in terms of
# priority.
sys.meta_path.append(SpamMetaPathFinder)

# Setting up a path entry finder.
# Make sure to put the path hook in the proper location in the list in terms
# of priority.
sys.path_hooks.append(SpamPathEntryFinder.path_hook(loader_details))
```

Approximating `importlib.import_module()`

Import itself is implemented in Python code, making it possible to expose most of the import machinery through importlib. The following helps illustrate the various APIs that importlib exposes by providing an approximate implementation of `importlib.import_module()` :

```
import importlib.util
import sys

def import_module(name, package=None):
    """An approximate implementation of import."""
    absolute_name = importlib.util.resolve_name(name, package)
    try:
        return sys.modules[absolute_name]
    except KeyError:
        pass

    path = None
    if '.' in absolute_name:
        parent_name, _, child_name = absolute_name.rpartition('.')
        parent_module = import_module(parent_name)
        path = parent_module.__spec__.submodule_search_locations
    for finder in sys.meta_path:
        spec = finder.find_spec(absolute_name, path)
        if spec is not None:
            break
    else:
        msg = f'No module named {absolute_name!r}'
        raise ModuleNotFoundError(msg, name=absolute_name)
    module = importlib.util.module_from_spec(spec)
```

(suite sur la page suivante)

(suite de la page précédente)

```
    sys.modules[absolute_name] = module
    spec.loader.exec_module(module)
    if path is not None:
        setattr(parent_module, child_name, module)
    return module
```

32.6 importlib.resources -- Package resource reading, opening and access

Source code : Lib/importlib/resources/__init__.py

Ajouté dans la version 3.7.

This module leverages Python's import system to provide access to *resources* within *packages*.

"Resources" are file-like resources associated with a module or package in Python. The resources may be contained directly in a package, within a subdirectory contained in that package, or adjacent to modules outside a package. Resources may be text or binary. As a result, Python module sources (.py) of a package and compilation artifacts (pycache) are technically de-facto resources of that package. In practice, however, resources are primarily those non-Python artifacts exposed specifically by the package author.

Resources can be opened or read in either binary or text mode.

Resources are roughly akin to files inside directories, though it's important to keep in mind that this is just a metaphor. Resources and packages **do not** have to exist as physical files and directories on the file system : for example, a package and its resources can be imported from a zip file using `zipimport`.

> **Note**
>
> This module provides functionality similar to pkg_resources Basic Resource Access without the performance overhead of that package. This makes reading resources included in packages easier, with more stable and consistent semantics.
>
> The standalone backport of this module provides more information on using importlib.resources and migrating from pkg_resources to importlib.resources.

`Loaders` that wish to support resource reading should implement a `get_resource_reader(fullname)` method as specified by `importlib.resources.abc.ResourceReader`.

class `importlib.resources.`**`Anchor`**

Represents an anchor for resources, either a `module object` or a module name as a string. Defined as `Union[str, ModuleType]`.

`importlib.resources.`**`files`**(*anchor :* Anchor | None = *None*)

Returns a `Traversable` object representing the resource container (think directory) and its resources (think files). A Traversable may contain other containers (think subdirectories).

anchor is an optional `Anchor`. If the anchor is a package, resources are resolved from that package. If a module, resources are resolved adjacent to that module (in the same package or the package root). If the anchor is omitted, the caller's module is used.

Ajouté dans la version 3.9.

Modifié dans la version 3.12 : *package* parameter was renamed to *anchor*. *anchor* can now be a non-package module and if omitted will default to the caller's module. *package* is still accepted for compatibility but will raise a `DeprecationWarning`. Consider passing the anchor positionally or using `importlib_resources >= 5.10` for a compatible interface on older Pythons.

importlib.resources.**as_file**(*traversable*)

Given a `Traversable` object representing a file or directory, typically from `importlib.resources.files()`, return a context manager for use in a `with` statement. The context manager provides a `pathlib.Path` object.

Exiting the context manager cleans up any temporary file or directory created when the resource was extracted from e.g. a zip file.

Use `as_file` when the Traversable methods (`read_text`, etc) are insufficient and an actual file or directory on the file system is required.

Ajouté dans la version 3.9.

Modifié dans la version 3.12 : Added support for *traversable* representing a directory.

32.6.1 API par fonction

A set of simplified, backwards-compatible helpers is available. These allow common operations in a single function call.

For all the following functions :

— *anchor* is an `Anchor`, as in `files()`. Unlike in `files`, it may not be omitted.

— *path_names* are components of a resource's path name, relative to the anchor. For example, to get the text of resource named `info.txt`, use :

```
importlib.resources.read_text(my_module, "info.txt")
```

Like `Traversable.joinpath`, The individual components should use forward slashes (`/`) as path separators. For example, the following are equivalent :

```
importlib.resources.read_binary(my_module, "pics/painting.png")
importlib.resources.read_binary(my_module, "pics", "painting.png")
```

For backward compatibility reasons, functions that read text require an explicit *encoding* argument if multiple *path_names* are given. For example, to get the text of `info/chapter1.txt`, use :

```
importlib.resources.read_text(my_module, "info", "chapter1.txt",
                              encoding='utf-8')
```

importlib.resources.**open_binary**(*anchor*, **path_names*)

Open the named resource for binary reading.

See *the introduction* for details on *anchor* and *path_names*.

This function returns a `BinaryIO` object, that is, a binary stream open for reading.

This function is roughly equivalent to :

```
files(anchor).joinpath(*path_names).open('rb')
```

Modifié dans la version 3.13 : Multiple *path_names* are accepted.

importlib.resources.**open_text**(*anchor*, **path_names*, *encoding='utf-8'*, *errors='strict'*)

Open the named resource for text reading. By default, the contents are read as strict UTF-8.

See *the introduction* for details on *anchor* and *path_names*. *encoding* and *errors* have the same meaning as in built-in `open()`.

For backward compatibility reasons, the *encoding* argument must be given explicitly if there are multiple *path_names*. This limitation is scheduled to be removed in Python 3.15.

This function returns a `TextIO` object, that is, a text stream open for reading.

This function is roughly equivalent to :

```
files(anchor).joinpath(*path_names).open('r', encoding=encoding)
```

Modifié dans la version 3.13 : Multiple *path_names* are accepted. *encoding* and *errors* must be given as keyword arguments.

`importlib.resources.`**`read_binary`**(*anchor*, **path_names*)

Read and return the contents of the named resource as `bytes`.

See *the introduction* for details on *anchor* and *path_names*.

This function is roughly equivalent to :

```
files(anchor).joinpath(*path_names).read_bytes()
```

Modifié dans la version 3.13 : Multiple *path_names* are accepted.

`importlib.resources.`**`read_text`**(*anchor*, **path_names*, *encoding='utf-8'*, *errors='strict'*)

Read and return the contents of the named resource as `str`. By default, the contents are read as strict UTF-8.

See *the introduction* for details on *anchor* and *path_names*. *encoding* and *errors* have the same meaning as in built-in `open()`.

For backward compatibility reasons, the *encoding* argument must be given explicitly if there are multiple *path_names*. This limitation is scheduled to be removed in Python 3.15.

This function is roughly equivalent to :

```
files(anchor).joinpath(*path_names).read_text(encoding=encoding)
```

Modifié dans la version 3.13 : Multiple *path_names* are accepted. *encoding* and *errors* must be given as keyword arguments.

`importlib.resources.`**`path`**(*anchor*, **path_names*)

Provides the path to the *resource* as an actual file system path. This function returns a context manager for use in a `with` statement. The context manager provides a `pathlib.Path` object.

Exiting the context manager cleans up any temporary files created, e.g. when the resource needs to be extracted from a zip file.

For example, the `stat()` method requires an actual file system path ; it can be used like this :

```
with importlib.resources.path(anchor, "resource.txt") as fspath:
    result = fspath.stat()
```

See *the introduction* for details on *anchor* and *path_names*.

This function is roughly equivalent to :

```
as_file(files(anchor).joinpath(*path_names))
```

Modifié dans la version 3.13 : Multiple *path_names* are accepted. *encoding* and *errors* must be given as keyword arguments.

`importlib.resources.`**`is_resource`**(*anchor*, **path_names*)

Return `True` if the named resource exists, otherwise `False`. This function does not consider directories to be resources.

See *the introduction* for details on *anchor* and *path_names*.

This function is roughly equivalent to :

```
files(anchor).joinpath(*path_names).is_file()
```

Modifié dans la version 3.13 : Multiple *path_names* are accepted.

`importlib.resources.contents(anchor, *path_names)`

Return an iterable over the named items within the package or path. The iterable returns names of resources (e.g. files) and non-resources (e.g. directories) as `str`. The iterable does not recurse into subdirectories.

See *the introduction* for details on *anchor* and *path_names*.

This function is roughly equivalent to :

```
for resource in files(anchor).joinpath(*path_names).iterdir():
    yield resource.name
```

Obsolète depuis la version 3.11 : Prefer `iterdir()` as above, which offers more control over the results and richer functionality.

32.7 `importlib.resources.abc` -- Abstract base classes for resources

Source code : Lib/importlib/resources/abc.py

Ajouté dans la version 3.11.

class `importlib.resources.abc.ResourceReader`

Superseded by TraversableResources

An *abstract base class* to provide the ability to read *resources*.

From the perspective of this ABC, a *resource* is a binary artifact that is shipped within a package. Typically this is something like a data file that lives next to the `__init__.py` file of the package. The purpose of this class is to help abstract out the accessing of such data files so that it does not matter if the package and its data file(s) are stored e.g. in a zip file versus on the file system.

For any of methods of this class, a *resource* argument is expected to be a *path-like object* which represents conceptually just a file name. This means that no subdirectory paths should be included in the *resource* argument. This is because the location of the package the reader is for, acts as the "directory". Hence the metaphor for directories and file names is packages and resources, respectively. This is also why instances of this class are expected to directly correlate to a specific package (instead of potentially representing multiple packages or a module).

Loaders that wish to support resource reading are expected to provide a method called `get_resource_reader(fullname)` which returns an object implementing this ABC's interface. If the module specified by fullname is not a package, this method should return `None`. An object compatible with this ABC should only be returned when the specified module is a package.

Obsolète depuis la version 3.12 : Use `importlib.resources.abc.TraversableResources` instead.

abstractmethod open_resource(*resource*)

Returns an opened, *file-like object* for binary reading of the *resource*.

Si la ressource ne peut pas être trouvée, `FileNotFoundError` est levée.

abstractmethod resource_path(*resource*)

Renvoie le chemin de *resource* dans le système de fichiers.

If the resource does not concretely exist on the file system, raise `FileNotFoundError`.

abstractmethod is_resource(*name*)

Returns `True` if the named *name* is considered a resource. `FileNotFoundError` is raised if *name* does not exist.

abstractmethod contents()

Returns an *iterable* of strings over the contents of the package. Do note that it is not required that

all names returned by the iterator be actual resources, e.g. it is acceptable to return names for which `is_resource()` would be false.

Allowing non-resource names to be returned is to allow for situations where how a package and its resources are stored are known a priori and the non-resource names would be useful. For instance, returning subdirectory names is allowed so that when it is known that the package and resources are stored on the file system then those subdirectory names can be used directly.

The abstract method returns an iterable of no items.

class `importlib.resources.abc.`**`Traversable`**

An object with a subset of `pathlib.Path` methods suitable for traversing directories and opening files.

For a representation of the object on the file-system, use `importlib.resources.as_file()`.

`name`

Abstract. The base name of this object without any parent references.

`abstractmethod iterdir`()

Yield Traversable objects in self.

`abstractmethod is_dir`()

Return `True` if self is a directory.

`abstractmethod is_file`()

Return `True` if self is a file.

`abstractmethod joinpath`(**pathsegments*)

Traverse directories according to *pathsegments* and return the result as `Traversable`.

Each *pathsegments* argument may contain multiple names separated by forward slashes (`/`, `posixpath.sep`). For example, the following are equivalent :

```
files.joinpath('subdir', 'subsuddir', 'file.txt')
files.joinpath('subdir/subsuddir/file.txt')
```

Note that some `Traversable` implementations might not be updated to the latest version of the protocol. For compatibility with such implementations, provide a single argument without path separators to each call to `joinpath`. For example :

```
files.joinpath('subdir').joinpath('subsubdir').joinpath('file.txt')
```

Modifié dans la version 3.11 : `joinpath` accepts multiple *pathsegments*, and these segments may contain forward slashes as path separators. Previously, only a single *child* argument was accepted.

`abstractmethod __truediv__`(*child*)

Return Traversable child in self. Equivalent to `joinpath(child)`.

`abstractmethod open`(*mode='r'*, **args*, ***kwargs*)

mode may be 'r' or 'rb' to open as text or binary. Return a handle suitable for reading (same as `pathlib.Path.open`).

When opening as text, accepts encoding parameters such as those accepted by `io.TextIOWrapper`.

`read_bytes`()

Read contents of self as bytes.

`read_text`(*encoding=None*)

Read contents of self as text.

class `importlib.resources.abc.`**`TraversableResources`**

An abstract base class for resource readers capable of serving the `importlib.resources.files()` interface. Subclasses `ResourceReader` and provides concrete implementations of the `ResourceReader`'s abstract methods. Therefore, any loader supplying `TraversableResources` also supplies `ResourceReader`.

Loaders that wish to support resource reading are expected to implement this interface.

abstractmethod files()

Returns a *importlib.resources.abc.Traversable* object for the loaded package.

32.8 importlib.metadata -- Accessing package metadata

Ajouté dans la version 3.8.

Modifié dans la version 3.10 : importlib.metadata is no longer provisional.

Source code : Lib/importlib/metadata/__init__.py

importlib.metadata is a library that provides access to the metadata of an installed Distribution Package, such as its entry points or its top-level names (Import Packages, modules, if any). Built in part on Python's import system, this library intends to replace similar functionality in the entry point API and metadata API of pkg_resources. Along with *importlib.resources*, this package can eliminate the need to use the older and less efficient pkg_resources package.

importlib.metadata operates on third-party *distribution packages* installed into Python's site-packages directory via tools such as pip. Specifically, it works with distributions with discoverable dist-info or egg-info directories, and metadata defined by the Core metadata specifications.

Important

These are *not* necessarily equivalent to or correspond 1 :1 with the top-level *import package* names that can be imported inside Python code. One *distribution package* can contain multiple *import packages* (and single modules), and one top-level *import package* may map to multiple *distribution packages* if it is a namespace package. You can use *packages_distributions()* to get a mapping between them.

By default, distribution metadata can live on the file system or in zip archives on *sys.path*. Through an extension mechanism, the metadata can live almost anywhere.

Voir aussi

https://importlib-metadata.readthedocs.io/

The documentation for importlib_metadata, which supplies a backport of importlib.metadata. This includes an API reference for this module's classes and functions, as well as a migration guide for existing users of pkg_resources.

32.8.1 Aperçu

Let's say you wanted to get the version string for a Distribution Package you've installed using pip. We start by creating a virtual environment and installing something into it :

```
$ python -m venv example
$ source example/bin/activate
(example) $ python -m pip install wheel
```

You can get the version string for wheel by running the following :

```
(example) $ python
>>> from importlib.metadata import version
>>> version('wheel')
'0.32.3'
```

You can also get a collection of entry points selectable by properties of the EntryPoint (typically 'group' or 'name'), such as `console_scripts`, `distutils.commands` and others. Each group contains a collection of *EntryPoint* objects.

You can get the *metadata for a distribution* :

```
>>> list(metadata('wheel'))
['Metadata-Version', 'Name', 'Version', 'Summary', 'Home-page', 'Author', 'Author-
↪email', 'Maintainer', 'Maintainer-email', 'License', 'Project-URL', 'Project-URL
↪', 'Project-URL', 'Keywords', 'Platform', 'Classifier', 'Classifier', 'Classifier
↪', 'Classifier', 'Classifier', 'Classifier', 'Classifier', 'Classifier',
↪'Classifier', 'Classifier', 'Classifier', 'Classifier', 'Requires-Python',
↪'Provides-Extra', 'Requires-Dist', 'Requires-Dist']
```

You can also get a *distribution's version number*, list its *constituent files*, and get a list of the distribution's *Distribution requirements*.

exception `importlib.metadata.`**`PackageNotFoundError`**

Subclass of `ModuleNotFoundError` raised by several functions in this module when queried for a distribution package which is not installed in the current Python environment.

32.8.2 API par fonction

This package provides the following functionality via its public API.

Entry points

`importlib.metadata.`**`entry_points`**(***select_params*)

Returns a `EntryPoints` instance describing entry points for the current environment. Any given keyword parameters are passed to the `select()` method for comparison to the attributes of the individual entry point definitions.

Note : it is not currently possible to query for entry points based on their `EntryPoint.dist` attribute (as different `Distribution` instances do not currently compare equal, even if they have the same attributes)

class `importlib.metadata.`**`EntryPoints`**

Details of a collection of installed entry points.

Also provides a `.groups` attribute that reports all identified entry point groups, and a `.names` attribute that reports all identified entry point names.

class `importlib.metadata.`**`EntryPoint`**

Details of an installed entry point.

Each `EntryPoint` instance has `.name`, `.group`, and `.value` attributes and a `.load()` method to resolve the value. There are also `.module`, `.attr`, and `.extras` attributes for getting the components of the `.value` attribute, and `.dist` for obtaining information regarding the distribution package that provides the entry point.

Query all entry points :

```
>>> eps = entry_points()
```

The `entry_points()` function returns a `EntryPoints` object, a collection of all `EntryPoint` objects with `names` and `groups` attributes for convenience :

```
>>> sorted(eps.groups)
['console_scripts', 'distutils.commands', 'distutils.setup_keywords', 'egg_info.
↪writers', 'setuptools.installation']
```

`EntryPoints` has a `select()` method to select entry points matching specific properties. Select entry points in the `console_scripts` group :

```
>>> scripts = eps.select(group='console_scripts')
```

Equivalently, since `entry_points()` passes keyword arguments through to select :

```
>>> scripts = entry_points(group='console_scripts')
```

Pick out a specific script named "wheel" (found in the wheel project) :

```
>>> 'wheel' in scripts.names
True
>>> wheel = scripts['wheel']
```

Equivalently, query for that entry point during selection :

```
>>> (wheel,) = entry_points(group='console_scripts', name='wheel')
>>> (wheel,) = entry_points().select(group='console_scripts', name='wheel')
```

Inspect the resolved entry point :

```
>>> wheel
EntryPoint(name='wheel', value='wheel.cli:main', group='console_scripts')
>>> wheel.module
'wheel.cli'
>>> wheel.attr
'main'
>>> wheel.extras
[]
>>> main = wheel.load()
>>> main
<function main at 0x103528488>
```

The `group` and `name` are arbitrary values defined by the package author and usually a client will wish to resolve all entry points for a particular group. Read the setuptools docs for more information on entry points, their definition, and usage.

Modifié dans la version 3.12 : The "selectable" entry points were introduced in `importlib_metadata` 3.6 and Python 3.10. Prior to those changes, `entry_points` accepted no parameters and always returned a dictionary of entry points, keyed by group. With `importlib_metadata` 5.0 and Python 3.12, `entry_points` always returns an `EntryPoints` object. See backports.entry_points_selectable for compatibility options.

Modifié dans la version 3.13 : `EntryPoint` objects no longer present a tuple-like interface (`__getitem__()`).

Distribution metadata

`importlib.metadata.`**`metadata`**(*distribution_name*)

Return the distribution metadata corresponding to the named distribution package as a *`PackageMetadata`* instance.

Raises *`PackageNotFoundError`* if the named distribution package is not installed in the current Python environment.

`class` `importlib.metadata.`**`PackageMetadata`**

A concrete implementation of the PackageMetadata protocol.

In addition to providing the defined protocol methods and attributes, subscripting the instance is equivalent to calling the `get()` method.

Every Distribution Package includes some metadata, which you can extract using the `metadata()` function :

```
>>> wheel_metadata = metadata('wheel')
```

The keys of the returned data structure name the metadata keywords, and the values are returned unparsed from the distribution metadata :

```
>>> wheel_metadata['Requires-Python']
'>=2.7, !=3.0.*, !=3.1.*, !=3.2.*, !=3.3.*'
```

PackageMetadata also presents a `json` attribute that returns all the metadata in a JSON-compatible form per **PEP 566** :

```
>>> wheel_metadata.json['requires_python']
'>=2.7, !=3.0.*, !=3.1.*, !=3.2.*, !=3.3.*'
```

The full set of available metadata is not described here. See the PyPA Core metadata specification for additional details.

Modifié dans la version 3.10 : The `Description` is now included in the metadata when presented through the payload. Line continuation characters have been removed.

The `json` attribute was added.

Distribution versions

`importlib.metadata.`**`version`**(*distribution_name*)

Return the installed distribution package version for the named distribution package.

Raises *PackageNotFoundError* if the named distribution package is not installed in the current Python environment.

The `version()` function is the quickest way to get a Distribution Package's version number, as a string :

```
>>> version('wheel')
'0.32.3'
```

Distribution files

`importlib.metadata.`**`files`**(*distribution_name*)

Return the full set of files contained within the named distribution package.

Raises *PackageNotFoundError* if the named distribution package is not installed in the current Python environment.

Returns *None* if the distribution is found but the installation database records reporting the files associated with the distribuion package are missing.

`class` `importlib.metadata.`**`PackagePath`**

A *pathlib.PurePath* derived object with additional `dist`, `size`, and `hash` properties corresponding to the distribution package's installation metadata for that file.

The `files()` function takes a Distribution Package name and returns all of the files installed by this distribution. Each file is reported as a *PackagePath* instance. For example :

```
>>> util = [p for p in files('wheel') if 'util.py' in str(p)][0]
>>> util
PackagePath('wheel/util.py')
>>> util.size
859
>>> util.dist
<importlib.metadata._hooks.PathDistribution object at 0x101e0cef0>
>>> util.hash
<FileHash mode: sha256 value: bYkw5oMccfazVCoYQwKkkemoVyMAFoR34mmKBx8R1NI>
```

Once you have the file, you can also read its contents :

```
>>> print(util.read_text())
import base64
import sys
...
def as_bytes(s):
    if isinstance(s, text_type):
        return s.encode('utf-8')
    return s
```

You can also use the `locate()` method to get the absolute path to the file :

```
>>> util.locate()
PosixPath('/home/gustav/example/lib/site-packages/wheel/util.py')
```

In the case where the metadata file listing files (`RECORD` or `SOURCES.txt`) is missing, `files()` will return *None*. The caller may wish to wrap calls to `files()` in always_iterable or otherwise guard against this condition if the target distribution is not known to have the metadata present.

Distribution requirements

`importlib.metadata.`**`requires`**(*distribution_name*)

Return the declared dependency specifiers for the named distribution package.

Raises *PackageNotFoundError* if the named distribution package is not installed in the current Python environment.

To get the full set of requirements for a Distribution Package, use the `requires()` function :

```
>>> requires('wheel')
["pytest (>=3.0.0) ; extra == 'test'", "pytest-cov ; extra == 'test'"]
```

Mapping import to distribution packages

`importlib.metadata.`**`packages_distributions`**()

Return a mapping from the top level module and import package names found via *sys.meta_path* to the names of the distribution packages (if any) that provide the corresponding files.

To allow for namespace packages (which may have members provided by multiple distribution packages), each top level import name maps to a list of distribution names rather than mapping directly to a single name.

A convenience method to resolve the Distribution Package name (or names, in the case of a namespace package) that provide each importable top-level Python module or Import Package :

```
>>> packages_distributions()
{'importlib_metadata': ['importlib-metadata'], 'yaml': ['PyYAML'], 'jaraco': [
↪'jaraco.classes', 'jaraco.functools'], ...}
```

Some editable installs, do not supply top-level names, and thus this function is not reliable with such installs.

Ajouté dans la version 3.10.

32.8.3 Distributions

`importlib.metadata.`**`distribution`**(*distribution_name*)

Return a *Distribution* instance describing the named distribution package.

Raises *PackageNotFoundError* if the named distribution package is not installed in the current Python environment.

class importlib.metadata.**Distribution**

Details of an installed distribution package.

Note : different `Distribution` instances do not currently compare equal, even if they relate to the same installed distribution and accordingly have the same attributes.

While the module level API described above is the most common and convenient usage, you can get all of that information from the `Distribution` class. `Distribution` is an abstract object that represents the metadata for a Python Distribution Package. You can get the concreate `Distribution` subclass instance for an installed distribution package by calling the *distribution()* function :

```
>>> from importlib.metadata import distribution
>>> dist = distribution('wheel')
>>> type(dist)
<class 'importlib.metadata.PathDistribution'>
```

Thus, an alternative way to get the version number is through the `Distribution` instance :

```
>>> dist.version
'0.32.3'
```

There are all kinds of additional metadata available on `Distribution` instances :

```
>>> dist.metadata['Requires-Python']
'>=2.7, !=3.0.*, !=3.1.*, !=3.2.*, !=3.3.*'
>>> dist.metadata['License']
'MIT'
```

For editable packages, an `origin` property may present **PEP 610** metadata :

```
>>> dist.origin.url
'file:///path/to/wheel-0.32.3.editable-py3-none-any.whl'
```

The full set of available metadata is not described here. See the PyPA Core metadata specification for additional details.

Ajouté dans la version 3.13 : The `.origin` property was added.

32.8.4 Distribution Discovery

By default, this package provides built-in support for discovery of metadata for file system and zip file Distribution Packages. This metadata finder search defaults to `sys.path`, but varies slightly in how it interprets those values from how other import machinery does. In particular :

- `importlib.metadata` does not honor *bytes* objects on `sys.path`.
- `importlib.metadata` will incidentally honor *pathlib.Path* objects on `sys.path` even though such values will be ignored for imports.

32.8.5 Implementing Custom Providers

`importlib.metadata` address two API surfaces, one for *consumers* and another for *providers*. Most users are consumers, consuming metadata provided by the packages. There are other use-cases, however, where users wish to expose metadata through some other mechanism, such as alongside a custom importer. Such a use case calls for a *custom provider*.

Because Distribution Package metadata is not available through *sys.path* searches, or package loaders directly, the metadata for a distribution is found through import system finders. To find a distribution package's metadata, `importlib.metadata` queries the list of *meta path finders* on *sys.meta_path*.

The implementation has hooks integrated into the `PathFinder`, serving metadata for distribution packages found on the file system.

The abstract class *importlib.abc.MetaPathFinder* defines the interface expected of finders by Python's import system. `importlib.metadata` extends this protocol by looking for an optional `find_distributions` callable on the finders from *sys.meta_path* and presents this extended interface as the `DistributionFinder` abstract base class, which defines this abstract method :

```
@abc.abstractmethod
def find_distributions(context=DistributionFinder.Context()) ->␣
↪Iterable[Distribution]:
    """Return an iterable of all Distribution instances capable of
    loading the metadata for packages for the indicated ``context``.
    """
```

The `DistributionFinder.Context` object provides `.path` and `.name` properties indicating the path to search and name to match and may supply other relevant context sought by the consumer.

In practice, to support finding distribution package metadata in locations other than the file system, subclass `Distribution` and implement the abstract methods. Then from a custom finder, return instances of this derived `Distribution` in the `find_distributions()` method.

Exemple

Imagine a custom finder that loads Python modules from a database :

```
class DatabaseImporter(importlib.abc.MetaPathFinder):
    def __init__(self, db):
        self.db = db

    def find_spec(self, fullname, target=None) -> ModuleSpec:
        return self.db.spec_from_name(fullname)

sys.meta_path.append(DatabaseImporter(connect_db(...)))
```

That importer now presumably provides importable modules from a database, but it provides no metadata or entry points. For this custom importer to provide metadata, it would also need to implement `DistributionFinder` :

```
from importlib.metadata import DistributionFinder

class DatabaseImporter(DistributionFinder):
    ...

    def find_distributions(self, context=DistributionFinder.Context()):
        query = dict(name=context.name) if context.name else {}
        for dist_record in self.db.query_distributions(query):
            yield DatabaseDistribution(dist_record)
```

In this way, `query_distributions` would return records for each distribution served by the database matching the query. For example, if `requests-1.0` is in the database, `find_distributions` would yield a `DatabaseDistribution` for `Context(name='requests')` or `Context(name=None)`.

For the sake of simplicity, this example ignores `context.path`. The `path` attribute defaults to `sys.path` and is the set of import paths to be considered in the search. A `DatabaseImporter` could potentially function without any concern for a search path. Assuming the importer does no partitioning, the "path" would be irrelevant. In order to illustrate the purpose of `path`, the example would need to illustrate a more complex `DatabaseImporter` whose behavior varied depending on `sys.path`/`PYTHONPATH`. In that case, the `find_distributions` should honor the `context.path` and only yield `Distributions` pertinent to that path.

`DatabaseDistribution`, then, would look something like :

```
class DatabaseDistribution(importlib.metadata.Distribution):
    def __init__(self, record):
```

(suite sur la page suivante)

(suite de la page précédente)

```
        self.record = record

    def read_text(self, filename):
        """
        Read a file like "METADATA" for the current distribution.
        """
        if filename == "METADATA":
            return f"""Name: {self.record.name}
Version: {self.record.version}
"""
        if filename == "entry_points.txt":
            return "\n".join(
              f"""[{ep.group}]\n{ep.name}={ep.value}"""
              for ep in self.record.entry_points)

    def locate_file(self, path):
        raise RuntimeError("This distribution has no file system")
```

This basic implementation should provide metadata and entry points for packages served by the `DatabaseImporter`, assuming that the `record` supplies suitable `.name`, `.version`, and `.entry_points` attributes.

The `DatabaseDistribution` may also provide other metadata files, like `RECORD` (required for `Distribution.files`) or override the implementation of `Distribution.files`. See the source for more inspiration.

32.9 The initialization of the `sys.path` module search path

A module search path is initialized when Python starts. This module search path may be accessed at *`sys.path`*.

The first entry in the module search path is the directory that contains the input script, if there is one. Otherwise, the first entry is the current directory, which is the case when executing the interactive shell, a `-c` command, or `-m` module.

The `PYTHONPATH` environment variable is often used to add directories to the search path. If this environment variable is found then the contents are added to the module search path.

> **Note**
>
> `PYTHONPATH` will affect all installed Python versions/environments. Be wary of setting this in your shell profile or global environment variables. The *`site`* module offers more nuanced techniques as mentioned below.

The next items added are the directories containing standard Python modules as well as any *extension modules* that these modules depend on. Extension modules are `.pyd` files on Windows and `.so` files on other platforms. The directory with the platform-independent Python modules is called `prefix`. The directory with the extension modules is called `exec_prefix`.

The `PYTHONHOME` environment variable may be used to set the `prefix` and `exec_prefix` locations. Otherwise these directories are found by using the Python executable as a starting point and then looking for various 'landmark' files and directories. Note that any symbolic links are followed so the real Python executable location is used as the search starting point. The Python executable location is called `home`.

Once `home` is determined, the `prefix` directory is found by first looking for `python`*`majorversionminorversion`*`.zip` (`python311.zip`). On Windows the zip archive is searched for in `home` and on Unix the archive is expected to be in `lib`. Note that the expected zip archive location is added to the module search path even if the archive does not exist. If no archive was found, Python on Windows will continue the search for `prefix` by looking for `Lib\os.py`. Python on Unix will look for `lib/python`*`majorversion.minorversion`*`/os.py` (`lib/python3.11/os.py`). On Windows `prefix` and `exec_prefix` are the same,

however on other platforms `lib/pythonmajorversion.minorversion/lib-dynload` (`lib/python3.11/lib-dynload`) is searched for and used as an anchor for `exec_prefix`. On some platforms `lib` may be `lib64` or another value, see `sys.platlibdir` and `PYTHONPLATLIBDIR`.

Once found, `prefix` and `exec_prefix` are available at `sys.base_prefix` and `sys.base_exec_prefix` respectively.

If `PYTHONHOME` is not set, and a `pyvenv.cfg` file is found alongside the main executable, or in its parent directory, `sys.prefix` and `sys.exec_prefix` get set to the directory containing `pyvenv.cfg`, otherwise they are set to the same value as `sys.base_prefix` and `sys.base_exec_prefix`, respectively. This is used by *Virtual Environments*.

Finally, the `site` module is processed and `site-packages` directories are added to the module search path. A common way to customize the search path is to create `sitecustomize` or `usercustomize` modules as described in the `site` module documentation.

> **Note**
>
> Certain command line options may further affect path calculations. See `-E`, `-I`, `-s` and `-S` for further details.

Modifié dans la version 3.14 : `sys.prefix` and `sys.exec_prefix` are now set to the `pyvenv.cfg` directory during the path initialization. This was previously done by `site`, therefore affected by `-S`.

32.9.1 Virtual Environments

Virtual environments place a `pyvenv.cfg` file in their prefix, which causes `sys.prefix` and `sys.exec_prefix` to point to them, instead of the base installation.

The `prefix` and `exec_prefix` values of the base installation are available at `sys.base_prefix` and `sys.base_exec_prefix`.

As well as being used as a marker to identify virtual environments, `pyvenv.cfg` may also be used to configure the `site` initialization. Please refer to `site`'s *virtual environments documentation*.

> **Note**
>
> `PYTHONHOME` overrides the `pyvenv.cfg` detection.

> **Note**
>
> There are other ways how "virtual environments" could be implemented, this documentation referes implementations based on the `pyvenv.cfg` mechanism, such as `venv`. Most virtual environment implementations follow the model set by `venv`, but there may be exotic implementations that diverge from it.

32.9.2 _pth files

To completely override `sys.path` create a `._pth` file with the same name as the shared library or executable (`python._pth` or `python311._pth`). The shared library path is always known on Windows, however it may not be available on other platforms. In the `._pth` file specify one line for each path to add to `sys.path`. The file based on the shared library name overrides the one based on the executable, which allows paths to be restricted for any program loading the runtime if desired.

Lorsque le fichier existe, toutes les variables de registre et d'environnement sont ignorées, le mode isolé est activé et `site` n'est pas importé, sauf si une ligne du fichier spécifie `import site`. Les chemins et les lignes vides commençant par # sont ignorés. Chaque chemin d'accès peut être absolu ou relatif à l'emplacement du fichier. Les importations autres que vers *site* ne sont pas autorisées, ni n'importe quelle instruction Python.

Notez que les fichiers `.pth` (sans signe souligné précédant le `pth`) seront traités normalement par le module *site* lorsque `import site` a été spécifié.

32.9.3 Embedded Python

If Python is embedded within another application `Py_InitializeFromConfig()` and the `PyConfig` structure can be used to initialize Python. The path specific details are described at init-path-config.

Voir aussi

— windows_finding_modules for detailed Windows notes.
— using-on-unix for Unix details.

CHAPITRE 33

Services du Langage Python

Python fournit quelques modules pour vous aider à travailler avec le langage Python lui-même. Ces modules gèrent entre autres l'analyse lexicale, l'analyse syntaxique, et le désassemblage de *bytecode*.

Ces modules sont :

33.1 `ast` --- Abstract Syntax Trees

Code source : Lib/ast.py

Le module `ast` permet aux applications Python de traiter les arbres syntaxiques, dérivés directement de la grammaire abstraite du langage. On peut notamment déterminer dans un programme la forme de chaque élément de grammaire, qui est susceptible d'être modifiée par les nouvelles versions de Python.

Un arbre syntaxique abstrait peut être généré en passant l'option `ast.PyCF_ONLY_AST` à la fonction native `compile()`, ou à l'aide de la fonction auxiliaire `parse()` fournie par ce module. Le résultat est un arbre composé d'objets dont les classes héritent toutes de `ast.AST`. On peut compiler les arbres en code objet Python à l'aide de la fonction native `compile()`.

33.1.1 Grammaire abstraite

La grammaire abstraite est actuellement définie comme suit :

```
-- ASDL's 4 builtin types are:
-- identifier, int, string, constant

module Python
{
    mod = Module(stmt* body, type_ignore* type_ignores)
        | Interactive(stmt* body)
        | Expression(expr body)
        | FunctionType(expr* argtypes, expr returns)

    stmt = FunctionDef(identifier name, arguments args,
                       stmt* body, expr* decorator_list, expr? returns,
                       string? type_comment, type_param* type_params)
```

(suite sur la page suivante)

(suite de la page précédente)

```
          | AsyncFunctionDef(identifier name, arguments args,
                            stmt* body, expr* decorator_list, expr? returns,
                            string? type_comment, type_param* type_params)

          | ClassDef(identifier name,
             expr* bases,
             keyword* keywords,
             stmt* body,
             expr* decorator_list,
             type_param* type_params)
          | Return(expr? value)

          | Delete(expr* targets)
          | Assign(expr* targets, expr value, string? type_comment)
          | TypeAlias(expr name, type_param* type_params, expr value)
          | AugAssign(expr target, operator op, expr value)
          -- 'simple' indicates that we annotate simple name without parens
          | AnnAssign(expr target, expr annotation, expr? value, int simple)

          -- use 'orelse' because else is a keyword in target languages
          | For(expr target, expr iter, stmt* body, stmt* orelse, string? type_
↪comment)
          | AsyncFor(expr target, expr iter, stmt* body, stmt* orelse, string?␣
↪type_comment)
          | While(expr test, stmt* body, stmt* orelse)
          | If(expr test, stmt* body, stmt* orelse)
          | With(withitem* items, stmt* body, string? type_comment)
          | AsyncWith(withitem* items, stmt* body, string? type_comment)

          | Match(expr subject, match_case* cases)

          | Raise(expr? exc, expr? cause)
          | Try(stmt* body, excepthandler* handlers, stmt* orelse, stmt* finalbody)
          | TryStar(stmt* body, excepthandler* handlers, stmt* orelse, stmt*␣
↪finalbody)
          | Assert(expr test, expr? msg)

          | Import(alias* names)
          | ImportFrom(identifier? module, alias* names, int? level)

          | Global(identifier* names)
          | Nonlocal(identifier* names)
          | Expr(expr value)
          | Pass | Break | Continue

          -- col_offset is the byte offset in the utf8 string the parser uses
          attributes (int lineno, int col_offset, int? end_lineno, int? end_col_
↪offset)

          -- BoolOp() can use left & right?
    expr = BoolOp(boolop op, expr* values)
         | NamedExpr(expr target, expr value)
         | BinOp(expr left, operator op, expr right)
         | UnaryOp(unaryop op, expr operand)
         | Lambda(arguments args, expr body)
         | IfExp(expr test, expr body, expr orelse)
```

(suite sur la page suivante)

(suite de la page précédente)

```
        | Dict(expr* keys, expr* values)
        | Set(expr* elts)
        | ListComp(expr elt, comprehension* generators)
        | SetComp(expr elt, comprehension* generators)
        | DictComp(expr key, expr value, comprehension* generators)
        | GeneratorExp(expr elt, comprehension* generators)
        -- the grammar constrains where yield expressions can occur
        | Await(expr value)
        | Yield(expr? value)
        | YieldFrom(expr value)
        -- need sequences for compare to distinguish between
        -- x < 4 < 3 and (x < 4) < 3
        | Compare(expr left, cmpop* ops, expr* comparators)
        | Call(expr func, expr* args, keyword* keywords)
        | FormattedValue(expr value, int conversion, expr? format_spec)
        | JoinedStr(expr* values)
        | Constant(constant value, string? kind)

        -- the following expression can appear in assignment context
        | Attribute(expr value, identifier attr, expr_context ctx)
        | Subscript(expr value, expr slice, expr_context ctx)
        | Starred(expr value, expr_context ctx)
        | Name(identifier id, expr_context ctx)
        | List(expr* elts, expr_context ctx)
        | Tuple(expr* elts, expr_context ctx)

        -- can appear only in Subscript
        | Slice(expr? lower, expr? upper, expr? step)

         -- col_offset is the byte offset in the utf8 string the parser uses
         attributes (int lineno, int col_offset, int? end_lineno, int? end_col_
↪offset)

   expr_context = Load | Store | Del

   boolop = And | Or

   operator = Add | Sub | Mult | MatMult | Div | Mod | Pow | LShift
                | RShift | BitOr | BitXor | BitAnd | FloorDiv

   unaryop = Invert | Not | UAdd | USub

   cmpop = Eq | NotEq | Lt | LtE | Gt | GtE | Is | IsNot | In | NotIn

   comprehension = (expr target, expr iter, expr* ifs, int is_async)

   excepthandler = ExceptHandler(expr? type, identifier? name, stmt* body)
                   attributes (int lineno, int col_offset, int? end_lineno, int?␣
↪end_col_offset)

   arguments = (arg* posonlyargs, arg* args, arg? vararg, arg* kwonlyargs,
                expr* kw_defaults, arg? kwarg, expr* defaults)

   arg = (identifier arg, expr? annotation, string? type_comment)
          attributes (int lineno, int col_offset, int? end_lineno, int? end_col_
↪offset)
```

(suite sur la page suivante)

(suite de la page précédente)

```
    -- keyword arguments supplied to call (NULL identifier for **kwargs)
    keyword = (identifier? arg, expr value)
              attributes (int lineno, int col_offset, int? end_lineno, int? end_
↪col_offset)

    -- import name with optional 'as' alias.
    alias = (identifier name, identifier? asname)
             attributes (int lineno, int col_offset, int? end_lineno, int? end_col_
↪offset)

    withitem = (expr context_expr, expr? optional_vars)

    match_case = (pattern pattern, expr? guard, stmt* body)

    pattern = MatchValue(expr value)
            | MatchSingleton(constant value)
            | MatchSequence(pattern* patterns)
            | MatchMapping(expr* keys, pattern* patterns, identifier? rest)
            | MatchClass(expr cls, pattern* patterns, identifier* kwd_attrs,␣
↪pattern* kwd_patterns)

            | MatchStar(identifier? name)
            -- The optional "rest" MatchMapping parameter handles capturing extra␣
↪mapping keys

            | MatchAs(pattern? pattern, identifier? name)
            | MatchOr(pattern* patterns)

             attributes (int lineno, int col_offset, int end_lineno, int end_col_
↪offset)

    type_ignore = TypeIgnore(int lineno, string tag)

    type_param = TypeVar(identifier name, expr? bound, expr? default_value)
               | ParamSpec(identifier name, expr? default_value)
               | TypeVarTuple(identifier name, expr? default_value)
              attributes (int lineno, int col_offset, int end_lineno, int end_col_
↪offset)
}
```

33.1.2 Classes de nœuds

class ast.**AST**

This is the base of all AST node classes. The actual node classes are derived from the `Parser/Python.asdl` file, which is reproduced *above*. They are defined in the `_ast` C module and re-exported in *ast*.

Il y a une classe définie pour chacun des symboles présents à gauche dans la grammaire abstraite (par exemple `ast.stmt` ou `ast.expr`). En plus de cela, une classe est définie pour chacun des constructeurs présentés à droite ; ces classes héritent des classes situées à leur gauche dans l'arbre. Par exemple, la classe *ast.BinOp* hérite de la classe `ast.expr`. Pour les règles de réécriture avec alternatives (les « sommes »), la classe de la partie gauche est abstraite : seules les classes de nœuds à droite sont instanciées.

_fields

Each concrete class has an attribute `_fields` which gives the names of all child nodes.

Chaque instance d'une classe concrète possède un attribut pour chaque nœud enfant, du type défini par

la grammaire. Par exemple, les instances *ast.BinOp* possèdent un attribut `left` de type `ast.expr`.

Si un attribut est marqué comme optionnel dans la grammaire (avec un point d'interrogation `?`), sa valeur peut être `None`. S'il peut avoir zéro, une ou plusieurs valeurs (ce qui est marqué par un astérisque `*`), elles sont regroupées dans une liste Python. Tous les attributs possibles doivent être présents et avoir une valeur valide pour compiler un arbre syntaxique avec *compile()*.

_field_types

The `_field_types` attribute on each concrete class is a dictionary mapping field names (as also listed in *_fields*) to their types.

```
>>> ast.TypeVar._field_types
{'name': <class 'str'>, 'bound': ast.expr | None, 'default_value': ast.
↪expr | None}
```

Ajouté dans la version 3.13.

lineno

col_offset

end_lineno

end_col_offset

Instances of `ast.expr` and `ast.stmt` subclasses have *lineno*, *col_offset*, *end_lineno*, and *end_col_offset* attributes. The *lineno* and *end_lineno* are the first and last line numbers of source text span (1-indexed so the first line is line 1) and the *col_offset* and *end_col_offset* are the corresponding UTF-8 byte offsets of the first and last tokens that generated the node. The UTF-8 offset is recorded because the parser uses UTF-8 internally.

Les décalages *end…* ne sont pas obligatoires ni nécessaires au compilateur. *end_col_offset* pointe *après* le dernier lexème. On peut donc obtenir la partie du code source ayant donné lieu à une expression avec `ligne_source[nœud.col_offset : nœud.end_col_offset]`.

Le constructeur d'une classe nommée `ast.T` analyse ses arguments comme suit :

— S'il y a des arguments positionnels, il doit y avoir autant de termes dans `T._fields`; ils sont assignés comme attributs portant ces noms.

— S'il y a des arguments nommés, ils définissent les attributs de mêmes noms avec les valeurs données.

Par exemple, pour créer et peupler un nœud *ast.UnaryOp*, on peut utiliser

```
node = ast.UnaryOp(ast.USub(), ast.Constant(5, lineno=0, col_offset=0),
                   lineno=0, col_offset=0)
```

If a field that is optional in the grammar is omitted from the constructor, it defaults to `None`. If a list field is omitted, it defaults to the empty list. If a field of type `ast.expr_context` is omitted, it defaults to *Load()*. If any other field is omitted, a *DeprecationWarning* is raised and the AST node will not have this field. In Python 3.15, this condition will raise an error.

Modifié dans la version 3.8 : toutes les constantes sont désormais représentées par des instances de *ast.Constant*.

Modifié dans la version 3.9 : Simple indices are represented by their value, extended slices are represented as tuples.

Modifié dans la version 3.14 : The `__repr__()` output of *AST* nodes includes the values of the node fields.

Obsolète depuis la version 3.8 : Old classes `ast.Num`, `ast.Str`, `ast.Bytes`, `ast.NameConstant` and `ast.Ellipsis` are still available, but they will be removed in future Python releases. In the meantime, instantiating them will return an instance of a different class.

Obsolète depuis la version 3.9 : Old classes `ast.Index` and `ast.ExtSlice` are still available, but they will be removed in future Python releases. In the meantime, instantiating them will return an instance of a different class.

Deprecated since version 3.13, will be removed in version 3.15 : Previous versions of Python allowed the creation of AST nodes that were missing required fields. Similarly, AST node constructors allowed arbitrary keyword arguments that were set as attributes of the AST node, even if they did not match any of the fields of the AST node. This behavior is deprecated and will be removed in Python 3.15.

> **Note**
>
> Les descriptions individuelles des classes de nœuds dans la prochaine section sont au départ adaptées du merveilleux projet Green Tree Snakes, porté par de nombreux contributeurs.

Root nodes

class ast.**Module**(*body*, *type_ignores*)

A Python module, as with file input. Node type generated by `ast.parse()` in the default `"exec"` *mode*.

`body` is a `list` of the module's *Instructions*.

`type_ignores` is a `list` of the module's type ignore comments ; see `ast.parse()` for more details.

```
>>> print(ast.dump(ast.parse('x = 1'), indent=4))
Module(
    body=[
        Assign(
            targets=[
                Name(id='x', ctx=Store())],
            value=Constant(value=1))])
```

class ast.**Expression**(*body*)

A single Python expression input. Node type generated by `ast.parse()` when *mode* is `"eval"`.

`body` is a single node, one of the *expression types*.

```
>>> print(ast.dump(ast.parse('123', mode='eval'), indent=4))
Expression(
    body=Constant(value=123))
```

class ast.**Interactive**(*body*)

A single interactive input, like in tut-interac. Node type generated by `ast.parse()` when *mode* is `"single"`.

`body` is a `list` of *statement nodes*.

```
>>> print(ast.dump(ast.parse('x = 1; y = 2', mode='single'), indent=4))
Interactive(
    body=[
        Assign(
            targets=[
                Name(id='x', ctx=Store())],
            value=Constant(value=1)),
        Assign(
            targets=[
                Name(id='y', ctx=Store())],
            value=Constant(value=2))])
```

class ast.**FunctionType**(*argtypes*, *returns*)

A representation of an old-style type comments for functions, as Python versions prior to 3.5 didn't support **PEP 484** annotations. Node type generated by `ast.parse()` when *mode* is `"func_type"`.

Such type comments would look like this :

```
def sum_two_number(a, b):
    # type: (int, int) -> int
    return a + b
```

`argtypes` is a `list` of *expression nodes*.

`returns` is a single *expression node*.

```
>>> print(ast.dump(ast.parse('(int, str) -> List[int]', mode='func_type'),␣
↪indent=4))
FunctionType(
    argtypes=[
        Name(id='int', ctx=Load()),
        Name(id='str', ctx=Load())],
    returns=Subscript(
        value=Name(id='List', ctx=Load()),
        slice=Name(id='int', ctx=Load()),
        ctx=Load()))
```

Ajouté dans la version 3.8.

Littéraux

class ast.**Constant**(*value*)

Valeur constante, contenue dans le champ *value*. Les valeurs possibles sont celles de types simples comme les nombres, les chaînes de caractères, ou encore `None`, mais aussi certains conteneurs immuables (*n*-uplets et ensembles figés) lorsque tous leurs éléments sont constants.

```
>>> print(ast.dump(ast.parse('123', mode='eval'), indent=4))
Expression(
    body=Constant(value=123))
```

class ast.**FormattedValue**(*value*, *conversion*, *format_spec*)

Champ de formatage dans une chaîne littérale formatée (*f-string*). Ce nœud peut être isolé si la chaîne contient un unique champ et rien d'autre. Sinon, il apparaît dans `JoinedStr`.

- *value* est un nœud d'expression quelconque (comme un littéral, une variable, ou encore un appel de fonction).
- *conversion* est un entier parmi les valeurs suivantes :
  - -1, aucun formatage ;
  - 115, pour le formatage par `str()` correspondant à `!s` ;
  - 114, pour le formatage par `repr()` correspondant à `!r` ;
  - 97, pour le formatage par `ascii()` correspondant à `!a`.
- *format_spec* est un nœud `JoinedStr` qui précise la manière de formater la valeur. Si aucun formatage particulier n'a été donné, *format_spec* vaut `None`. *conversion* et *format_spec* peuvent tout à fait coexister.

class ast.**JoinedStr**(*values*)

Chaîne littérale formatée (*f-string*), qui contient une liste de nœuds `FormattedValue` et `Constant`.

```
>>> print(ast.dump(ast.parse('f"sin({a}) is {sin(a):.3}"', mode='eval'),␣
↪indent=4))
Expression(
    body=JoinedStr(
        values=[
            Constant(value='sin('),
            FormattedValue(
                value=Name(id='a', ctx=Load()),
                conversion=-1),
            Constant(value=') is '),
            FormattedValue(
                value=Call(
                    func=Name(id='sin', ctx=Load()),
                    args=[
                        Name(id='a', ctx=Load())]),
                conversion=-1,
                format_spec=JoinedStr(
```

(suite sur la page suivante)

(suite de la page précédente)

```
                    values=[
                        Constant(value='.3')]))]))
```

class ast.**List**(*elts*, *ctx*)

class ast.**Tuple**(*elts*, *ctx*)

Liste ou *n*-uplet, dont les éléments sont rassemblés dans la liste *elts*. *ctx* est une instance de `Store` si la liste ou le *n*-uplet est la cible d'une affectation (par exemple `(x, y) = quelque_chose`). Sinon, c'est une instance de `Load`.

```
>>> print(ast.dump(ast.parse('[1, 2, 3]', mode='eval'), indent=4))
Expression(
    body=List(
        elts=[
            Constant(value=1),
            Constant(value=2),
            Constant(value=3)],
        ctx=Load()))
>>> print(ast.dump(ast.parse('(1, 2, 3)', mode='eval'), indent=4))
Expression(
    body=Tuple(
        elts=[
            Constant(value=1),
            Constant(value=2),
            Constant(value=3)],
        ctx=Load()))
```

class ast.**Set**(*elts*)

Ensemble. *elts* est la liste de ses éléments.

```
>>> print(ast.dump(ast.parse('{1, 2, 3}', mode='eval'), indent=4))
Expression(
    body=Set(
        elts=[
            Constant(value=1),
            Constant(value=2),
            Constant(value=3)]))
```

class ast.**Dict**(*keys*, *values*)

Dictionnaire. Les listes *keys* et *values* contiennent respectivement les clés et les valeurs. Leurs ordres se correspondent, c'est-à-dire que la valeur associée à l'élément d'indice *i* dans *keys* est à chercher à l'indice *i* de *values*. *keys* et *values* sont donc des équivalents à `dictionnaire.keys()` et `dictionnaire.values()`.

Si un dictionnaire littéral contient une expression doublement étoilée à déballer, elle est intégrée dans *values*, et `None` est mis à la place correspondante dans *keys*.

```
>>> print(ast.dump(ast.parse('{"a":1, **d}', mode='eval'), indent=4))
Expression(
    body=Dict(
        keys=[
            Constant(value='a'),
            None],
        values=[
            Constant(value=1),
            Name(id='d', ctx=Load())]))
```

Variables

class ast.**Name**(*id*, *ctx*)

Variable, dont le nom est *id* (une chaîne de caractères). *ctx* est de l'un des trois types :

class ast.**Load**

class ast.**Store**

class ast.**Del**

Ces types de contexte distinguent les variables selon qu'elles sont utilisées pour lire la valeur, mettre la variable à une nouvelle valeur, ou supprimer la variable.

```
>>> print(ast.dump(ast.parse('a'), indent=4))
Module(
    body=[
        Expr(
            value=Name(id='a', ctx=Load()))])

>>> print(ast.dump(ast.parse('a = 1'), indent=4))
Module(
    body=[
        Assign(
            targets=[
                Name(id='a', ctx=Store())],
            value=Constant(value=1))])

>>> print(ast.dump(ast.parse('del a'), indent=4))
Module(
    body=[
        Delete(
            targets=[
                Name(id='a', ctx=Del())])])
```

class ast.**Starred**(*value*, *ctx*)

Élément étoilé. *value* est le nœud auquel s'applique l'étoile. Le plus souvent, *value* est une instance de `Name`. Ce type est notamment nécessaire pour les appels de fonction avec déballage d'arguments (par exemple `fonction(*args)` ; voir aussi `Call`).

```
>>> print(ast.dump(ast.parse('a, *b = it'), indent=4))
Module(
    body=[
        Assign(
            targets=[
                Tuple(
                    elts=[
                        Name(id='a', ctx=Store()),
                        Starred(
                            value=Name(id='b', ctx=Store()),
                            ctx=Store())],
                    ctx=Store())],
            value=Name(id='it', ctx=Load()))])
```

Expressions

class ast.**Expr**(*value*)

Lorsque une expression, comme l'appel d'une fonction, apparaît comme une instruction à elle seule, sans que la valeur ne soit utilisée ou sauvegardée, l'expression est insérée dans ce conteneur. Le type de *value* peut être l'un des autres nœuds décrits dans cette section, ou bien parmi `Constant`, `Name`, `Lambda`, `Yield` et `YieldFrom`.

```
>>> print(ast.dump(ast.parse('-a'), indent=4))
Module(
    body=[
        Expr(
            value=UnaryOp(
                op=USub(),
                operand=Name(id='a', ctx=Load())))])
```

class ast.**UnaryOp**(*op*, *operand*)

Unité lexicale désignant une opération unaire. L'opérateur est *op*, l'opérande *operand* est un nœud d'expression quelconque.

class ast.**UAdd**

class ast.**USub**

class ast.**Not**

class ast.**Invert**

Unités lexicales désignant des opérations unaires. `Not` correspond au mot-clé `not`, `Invert` à l'opérateur `~`.

```
>>> print(ast.dump(ast.parse('not x', mode='eval'), indent=4))
Expression(
    body=UnaryOp(
        op=Not(),
        operand=Name(id='x', ctx=Load())))
```

class ast.**BinOp**(*left*, *op*, *right*)

Opération binaire (comme l'addition ou la division). L'opérateur est *op*, les opérandes *left* et *right* sont des nœuds d'expression quelconques.

```
>>> print(ast.dump(ast.parse('x + y', mode='eval'), indent=4))
Expression(
    body=BinOp(
        left=Name(id='x', ctx=Load()),
        op=Add(),
        right=Name(id='y', ctx=Load())))
```

class ast.**Add**

class ast.**Sub**

class ast.**Mult**

class ast.**Div**

class ast.**FloorDiv**

class ast.**Mod**

class ast.**Pow**

class ast.**LShift**

class ast.**RShift**

class ast.**BitOr**

class ast.**BitXor**

class ast.**BitAnd**

class ast.**MatMult**

Unités lexicales pour les opérations binaires.

class ast.**BoolOp**(*op*, *values*)

Opération booléenne, c'est-à-dire `and` ou `or`, entre *op* et les éléments de *values*. Les deux opérateurs sont distingués par le type de *op*, à savoir `And` ou `Or`. Les applications successives du même opérateur (comme `a or b or c`) sont regroupées dans un nœud unique avec plusieurs éléments dans la liste *values*.

L'opérateur `not` n'est pas implémenté ici, mais bien dans `UnaryOp`.

```
>>> print(ast.dump(ast.parse('x or y', mode='eval'), indent=4))
Expression(
    body=BoolOp(
        op=Or(),
        values=[
            Name(id='x', ctx=Load()),
            Name(id='y', ctx=Load())]))
```

class ast.**And**

class ast.**Or**

Unités lexicales pour les opérations booléennes.

class ast.**Compare**(*left*, *ops*, *comparators*)

Comparaison de deux valeurs ou plus. *left* est le premier élément de la comparaison, *ops* la liste des opérateurs, et *comparators* la liste des éléments restants de la comparaison.

```
>>> print(ast.dump(ast.parse('1 <= a < 10', mode='eval'), indent=4))
Expression(
    body=Compare(
        left=Constant(value=1),
        ops=[
            LtE(),
            Lt()],
        comparators=[
            Name(id='a', ctx=Load()),
            Constant(value=10)]))
```

class ast.**Eq**

class ast.**NotEq**

class ast.**Lt**

class ast.**LtE**

class ast.**Gt**

class ast.**GtE**

class ast.**Is**

class ast.**IsNot**

class ast.**In**

class ast.**NotIn**

Unités lexicales pour les comparaisons.

class ast.**Call**(*func*, *args*, *keywords*)

Appel d'une fonction. Le nœud *func*, représentant la fonction appelée, est habituellement de type `Name` ou `Attribute`. Les arguments sont contenus dans :

— *args*, la liste des arguments passés sans les nommer (arguments positionnels) ;

— `keywords` holds a list of `keyword` objects representing arguments passed by keyword.

The `args` and `keywords` arguments are optional and default to empty lists.

```
>>> print(ast.dump(ast.parse('func(a, b=c, *d, **e)', mode='eval'), indent=4))
Expression(
    body=Call(
        func=Name(id='func', ctx=Load()),
        args=[
            Name(id='a', ctx=Load()),
            Starred(
                value=Name(id='d', ctx=Load()),
                ctx=Load())],
```

(suite sur la page suivante)

(suite de la page précédente)

```
        keywords=[
            keyword(
                arg='b',
                value=Name(id='c', ctx=Load())),
            keyword(
                value=Name(id='e', ctx=Load()))]))
```

class ast.**keyword**(*arg*, *value*)

A keyword argument to a function call or class definition. `arg` is a raw string of the parameter name, `value` is a node to pass in.

class ast.**IfExp**(*test*, *body*, *orelse*)

Expression ternaire (`a if b else c`). Chaque champ contient un unique nœud. Dans l'exemple suivant, les trois sont de la classe `Name`.

```
>>> print(ast.dump(ast.parse('a if b else c', mode='eval'), indent=4))
Expression(
    body=IfExp(
        test=Name(id='b', ctx=Load()),
        body=Name(id='a', ctx=Load()),
        orelse=Name(id='c', ctx=Load())))
```

class ast.**Attribute**(*value*, *attr*, *ctx*)

Accès à un attribut (comme `d.keys`). *value* est un nœud, souvent de la classe `Name`. *attr* est le nom de l'attribut, sous forme de chaîne de caractères. *ctx* est de classe `Load`, `Store` ou `Del` selon le type de l'action effectuée sur l'attribut.

```
>>> print(ast.dump(ast.parse('snake.colour', mode='eval'), indent=4))
Expression(
    body=Attribute(
        value=Name(id='snake', ctx=Load()),
        attr='colour',
        ctx=Load()))
```

class ast.**NamedExpr**(*target*, *value*)

Expression d'affectation. Ces nœuds proviennent de l'opérateur « morse » `:=`. Contrairement à `Assign`, dont le premier argument est une liste qui peut contenir plusieurs cibles, les arguments *target* et *value* sont ici des nœuds simples.

```
>>> print(ast.dump(ast.parse('(x := 4)', mode='eval'), indent=4))
Expression(
    body=NamedExpr(
        target=Name(id='x', ctx=Store()),
        value=Constant(value=4)))
```

Ajouté dans la version 3.8.

Indiçage

class ast.**Subscript**(*value*, *slice*, *ctx*)

Accès à un indice, par exemple `liste[1]`. *value* est l'objet où l'indice est pris, donc la plupart du temps une séquence ou une table associative. *slice* est soit un indice, soit une tranche, soit une clé. Les instances de `Tuple` pour *slice* peuvent contenir des objets `Slice`. *ctx* est de type `Load`, `Store` ou `Del` selon l'action appliquée à l'indice.

```
>>> print(ast.dump(ast.parse('l[1:2, 3]', mode='eval'), indent=4))
Expression(
    body=Subscript(
        value=Name(id='l', ctx=Load()),
        slice=Tuple(
            elts=[
                Slice(
                    lower=Constant(value=1),
                    upper=Constant(value=2)),
                Constant(value=3)],
            ctx=Load()),
        ctx=Load()))
```

class ast.**Slice**(*lower*, *upper*, *step*)

Tranches normales (de la forme `début:fin` ou `début:fin:pas`). Les instances de cette classe ne peuvent apparaître que dans le champ *slice* d'un `Subscript`, que ce soit directement ou en tant qu'élément d'un `Tuple`.

```
>>> print(ast.dump(ast.parse('l[1:2]', mode='eval'), indent=4))
Expression(
    body=Subscript(
        value=Name(id='l', ctx=Load()),
        slice=Slice(
            lower=Constant(value=1),
            upper=Constant(value=2)),
        ctx=Load()))
```

Compréhensions

class ast.**ListComp**(*elt*, *generators*)

class ast.**SetComp**(*elt*, *generators*)

class ast.**GeneratorExp**(*elt*, *generators*)

class ast.**DictComp**(*key*, *value*, *generators*)

Liste, ensemble ou dictionnaire en compréhension, ou expression génératrice. *elt* est l'expression avant le premier `for`, évaluée à chaque itération. Il est remplacé par *key* (expression pour la clé) et *value* (expression pour la valeur) dans le cas des dictionnaires.

generators est une liste de nœuds `comprehension`.

```
>>> print(ast.dump(
...     ast.parse('[x for x in numbers]', mode='eval'),
...     indent=4,
... ))
Expression(
    body=ListComp(
        elt=Name(id='x', ctx=Load()),
        generators=[
            comprehension(
                target=Name(id='x', ctx=Store()),
                iter=Name(id='numbers', ctx=Load()),
                is_async=0)]))
>>> print(ast.dump(
...     ast.parse('{x: x**2 for x in numbers}', mode='eval'),
...     indent=4,
... ))
Expression(
    body=DictComp(
```

(suite sur la page suivante)

(suite de la page précédente)

```
        key=Name(id='x', ctx=Load()),
        value=BinOp(
            left=Name(id='x', ctx=Load()),
            op=Pow(),
            right=Constant(value=2)),
        generators=[
            comprehension(
                target=Name(id='x', ctx=Store()),
                iter=Name(id='numbers', ctx=Load()),
                is_async=0)]))
>>> print(ast.dump(
...     ast.parse('{x for x in numbers}', mode='eval'),
...     indent=4,
... ))
Expression(
    body=SetComp(
        elt=Name(id='x', ctx=Load()),
        generators=[
            comprehension(
                target=Name(id='x', ctx=Store()),
                iter=Name(id='numbers', ctx=Load()),
                is_async=0)]))
```

class ast.**comprehension**(*target*, *iter*, *ifs*, *is_async*)

Une clause `for` à l'intérieur d'une compréhension. *iter* est l'objet à parcourir, *target* est la cible de l'affectation qui se produit à chaque itération (la plupart du temps un nœud `Name` ou `Tuple`). *ifs* contient les tests qui décident si l'élément doit être inséré. C'est une liste, car chaque `for` peut être suivi de plusieurs `if`.

is_async est une valeur booléenne sous forme d'entier, 0 ou 1, qui indique si la compréhension est asynchrone, c'est-à-dire qu'elle a été écrite avec `async for` au lieu de `for`.

```
>>> print(ast.dump(ast.parse('[ord(c) for line in file for c in line]', mode=
↪'eval'),
...                indent=4)) # Multiple comprehensions in one.
Expression(
    body=ListComp(
        elt=Call(
            func=Name(id='ord', ctx=Load()),
            args=[
                Name(id='c', ctx=Load())]),
        generators=[
            comprehension(
                target=Name(id='line', ctx=Store()),
                iter=Name(id='file', ctx=Load()),
                is_async=0),
            comprehension(
                target=Name(id='c', ctx=Store()),
                iter=Name(id='line', ctx=Load()),
                is_async=0)]))

>>> print(ast.dump(ast.parse('(n**2 for n in it if n>5 if n<10)', mode='eval'),
...                indent=4)) # generator comprehension
Expression(
    body=GeneratorExp(
        elt=BinOp(
            left=Name(id='n', ctx=Load()),
            op=Pow(),
```

(suite sur la page suivante)

(suite de la page précédente)

```
            right=Constant(value=2)),
        generators=[
            comprehension(
                target=Name(id='n', ctx=Store()),
                iter=Name(id='it', ctx=Load()),
                ifs=[
                    Compare(
                        left=Name(id='n', ctx=Load()),
                        ops=[
                            Gt()],
                        comparators=[
                            Constant(value=5)]),
                    Compare(
                        left=Name(id='n', ctx=Load()),
                        ops=[
                            Lt()],
                        comparators=[
                            Constant(value=10)])],
                is_async=0)]))

>>> print(ast.dump(ast.parse('[i async for i in soc]', mode='eval'),
...                indent=4)) # Async comprehension
Expression(
    body=ListComp(
        elt=Name(id='i', ctx=Load()),
        generators=[
            comprehension(
                target=Name(id='i', ctx=Store()),
                iter=Name(id='soc', ctx=Load()),
                is_async=1)]))
```

Instructions

class ast.**Assign**(*targets*, *value*, *type_comment*)

Affectation. *targets* est une liste de nœuds, *value* est simplement un nœud.

S'il y a plusieurs nœuds dans *targets*, ils sont tous pris comme des cibles auxquelles est affectée la même expression. Le déballage de séquences est représenté par des nœuds `Tuple` ou `List` comme éléments de *targets*.

type_comment

Le champ facultatif *type_comment* contient une annotation de type fournie par un commentaire, et ce sous la forme d'une chaîne de caractères.

```
>>> print(ast.dump(ast.parse('a = b = 1'), indent=4)) # Multiple assignment
Module(
    body=[
        Assign(
            targets=[
                Name(id='a', ctx=Store()),
                Name(id='b', ctx=Store())],
            value=Constant(value=1))])

>>> print(ast.dump(ast.parse('a,b = c'), indent=4)) # Unpacking
Module(
    body=[
        Assign(
```

(suite sur la page suivante)

(suite de la page précédente)

```
            targets=[
                Tuple(
                    elts=[
                        Name(id='a', ctx=Store()),
                        Name(id='b', ctx=Store())],
                    ctx=Store())],
            value=Name(id='c', ctx=Load()))])
```

class ast.**AnnAssign**(*target*, *annotation*, *value*, *simple*)

An assignment with a type annotation. `target` is a single node and can be a *Name*, an *Attribute* or a *Subscript*. `annotation` is the annotation, such as a *Constant* or *Name* node. `value` is a single optional node.

`simple` is always either 0 (indicating a "complex" target) or 1 (indicating a "simple" target). A "simple" target consists solely of a *Name* node that does not appear between parentheses; all other targets are considered complex. Only simple targets appear in the `__annotations__` dictionary of modules and classes.

```
>>> print(ast.dump(ast.parse('c: int'), indent=4))
Module(
    body=[
        AnnAssign(
            target=Name(id='c', ctx=Store()),
            annotation=Name(id='int', ctx=Load()),
            simple=1)])

>>> print(ast.dump(ast.parse('(a): int = 1'), indent=4)) # Annotation with␣
↪parenthesis
Module(
    body=[
        AnnAssign(
            target=Name(id='a', ctx=Store()),
            annotation=Name(id='int', ctx=Load()),
            value=Constant(value=1),
            simple=0)])

>>> print(ast.dump(ast.parse('a.b: int'), indent=4)) # Attribute annotation
Module(
    body=[
        AnnAssign(
            target=Attribute(
                value=Name(id='a', ctx=Load()),
                attr='b',
                ctx=Store()),
            annotation=Name(id='int', ctx=Load()),
            simple=0)])

>>> print(ast.dump(ast.parse('a[1]: int'), indent=4)) # Subscript annotation
Module(
    body=[
        AnnAssign(
            target=Subscript(
                value=Name(id='a', ctx=Load()),
                slice=Constant(value=1),
                ctx=Store()),
            annotation=Name(id='int', ctx=Load()),
            simple=0)])
```

class ast.**AugAssign**(*target*, *op*, *value*)

Affectation incrémentale, par exemple `a += 1`. Dans l'exemple qui suit, *target* est un nœud de type `Name` avec le nom `'x'` (de contexte `Store`), *op* est une instance de `Add`, et *value* est un nœud `Constant` contenant la valeur 1.

The `target` attribute cannot be of class `Tuple` or `List`, unlike the targets of `Assign`.

```
>>> print(ast.dump(ast.parse('x += 2'), indent=4))
Module(
    body=[
        AugAssign(
            target=Name(id='x', ctx=Store()),
            op=Add(),
            value=Constant(value=2))])
```

class `ast.`**`Raise`**(*exc*, *cause*)

Instruction `raise`. *exc* est l'exception à lever, la plupart du temps un nœud de type `Call` ou `Name`. Ce champ vaut `None` dans le cas d'un `raise` sans rien d'autre. Le champ facultatif *cause* correspond à la partie après le `from` dans `raise ... from ...`.

```
>>> print(ast.dump(ast.parse('raise x from y'), indent=4))
Module(
    body=[
        Raise(
            exc=Name(id='x', ctx=Load()),
            cause=Name(id='y', ctx=Load()))])
```

class `ast.`**`Assert`**(*test*, *msg*)

Assertion. *test* est la condition, par exemple un nœud `Compare`. *msg* est le message d'erreur affiché si le test s'évalue comme faux.

```
>>> print(ast.dump(ast.parse('assert x,y'), indent=4))
Module(
    body=[
        Assert(
            test=Name(id='x', ctx=Load()),
            msg=Name(id='y', ctx=Load()))])
```

class `ast.`**`Delete`**(*targets*)

Instruction `del`. Les éléments à supprimer sont rassemblés dans la liste *target*. Ce sont généralement des nœuds de type `Name`, `Attribute` ou `Subscript`.

```
>>> print(ast.dump(ast.parse('del x,y,z'), indent=4))
Module(
    body=[
        Delete(
            targets=[
                Name(id='x', ctx=Del()),
                Name(id='y', ctx=Del()),
                Name(id='z', ctx=Del())])])
```

class `ast.`**`Pass`**

Instruction `pass`.

```
>>> print(ast.dump(ast.parse('pass'), indent=4))
Module(
    body=[
        Pass()])
```

class ast.**TypeAlias**(*name*, *type_params*, *value*)

A *type alias* created through the `type` statement. `name` is the name of the alias, `type_params` is a list of *type parameters*, and `value` is the value of the type alias.

```
>>> print(ast.dump(ast.parse('type Alias = int'), indent=4))
Module(
    body=[
        TypeAlias(
            name=Name(id='Alias', ctx=Store()),
            value=Name(id='int', ctx=Load()))])
```

Ajouté dans la version 3.12.

Il existe d'autres instructions, qui ne peuvent apparaître que dans certains contextes spécifiques comme l'intérieur d'une fonction ou d'une boucle. Elles sont décrites dans d'autres sections de ce document.

Importations

class ast.**Import**(*names*)

Instruction `import`. *names* prend la forme d'une liste de nœuds `alias`.

```
>>> print(ast.dump(ast.parse('import x,y,z'), indent=4))
Module(
    body=[
        Import(
            names=[
                alias(name='x'),
                alias(name='y'),
                alias(name='z')])])
```

class ast.**ImportFrom**(*module*, *names*, *level*)

Instruction `from ... import ...`. La partie après `from` est mise dans *module* comme une simple chaîne de caractères. Dans le cas des importations relatives, comme `from ..module import quelque_chose`, *module* ne contient pas les points au début. Dans les instructions de la forme `from . import toto`, *module* vaut `None`. *level* est le niveau d'importation relative sous forme d'entier (0 pour les importations absolues).

```
>>> print(ast.dump(ast.parse('from y import x,y,z'), indent=4))
Module(
    body=[
        ImportFrom(
            module='y',
            names=[
                alias(name='x'),
                alias(name='y'),
                alias(name='z')],
            level=0)])
```

class ast.**alias**(*name*, *asname*)

Correspond à `... as ...` dans une importation. *name* et *asname* sont de simples chaînes de caractères, qui correspondent aux deux parties de `... as ...`. Dans les instructions sans `as` (où le module n'est pas renommé), *asname* vaut `None`.

```
>>> print(ast.dump(ast.parse('from ..foo.bar import a as b, c'), indent=4))
Module(
    body=[
        ImportFrom(
            module='foo.bar',
            names=[
```

(suite sur la page suivante)

(suite de la page précédente)

```
                alias(name='a', asname='b'),
                alias(name='c')],
            level=2)])
```

Contrôle de l'exécution

> **Note**
>
> Si une clause facultative comme `else` est absente, le champ correspondant dans l'arbre est une liste vide.

class ast.**If**(*test*, *body*, *orelse*)

Instruction `if`. La condition, *test*, est un nœud, par exemple une instance de `Compare`. *body* est la liste des instructions du bloc. *orelse* contient les instructions dans le `else` (liste vide s'il n'y a pas de `else`).

Les clauses `elif` ne possèdent pas de représentation particulière. Elles sont simplement mises dans le *orelse* du `if` ou `elif` précédent, comme nœuds de type `If`.

```
>>> print(ast.dump(ast.parse("""
... if x:
...     ...
... elif y:
...     ...
... else:
...     ...
... """), indent=4))
Module(
    body=[
        If(
            test=Name(id='x', ctx=Load()),
            body=[
                Expr(
                    value=Constant(value=Ellipsis))],
            orelse=[
                If(
                    test=Name(id='y', ctx=Load()),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))],
                    orelse=[
                        Expr(
                            value=Constant(value=Ellipsis))])])])
```

class ast.**For**(*target*, *iter*, *body*, *orelse*, *type_comment*)

A `for` loop. `target` holds the variable(s) the loop assigns to, as a single `Name`, `Tuple`, `List`, `Attribute` or `Subscript` node. `iter` holds the item to be looped over, again as a single node. `body` and `orelse` contain lists of nodes to execute. Those in `orelse` are executed if the loop finishes normally, rather than via a `break` statement.

type_comment

Le champ facultatif *type_comment* contient une annotation de type fournie par un commentaire, et ce sous la forme d'une chaîne de caractères.

```
>>> print(ast.dump(ast.parse("""
... for x in y:
...     ...
```

(suite sur la page suivante)

(suite de la page précédente)

```
... else:
...     ...
... """), indent=4))
Module(
    body=[
        For(
            target=Name(id='x', ctx=Store()),
            iter=Name(id='y', ctx=Load()),
            body=[
                Expr(
                    value=Constant(value=Ellipsis))],
            orelse=[
                Expr(
                    value=Constant(value=Ellipsis))])])
```

class ast.**While**(*test*, *body*, *orelse*)

Boucle `while`. *test* est la condition évaluée à chaque itération, par exemple un nœud `Compare`. *body* contient les instructions du bloc, et *orelse* celles d'un éventuel bloc `else`, exécuté lorsque la boucle termine parce que la condition devient fausse et non pas à cause d'une instruction `break`.

```
>> print(ast.dump(ast.parse("""
... while x:
...     ...
... else:
...     ...
... """), indent=4))
Module(
    body=[
        While(
            test=Name(id='x', ctx=Load()),
            body=[
                Expr(
                    value=Constant(value=Ellipsis))],
            orelse=[
                Expr(
                    value=Constant(value=Ellipsis))])])
```

class ast.**Break**

class ast.**Continue**

Instructions `break` et `continue`.

```
>>> print(ast.dump(ast.parse("""\
... for a in b:
...     if a > 5:
...         break
...     else:
...         continue
...
... """), indent=4))
Module(
    body=[
        For(
            target=Name(id='a', ctx=Store()),
            iter=Name(id='b', ctx=Load()),
            body=[
                If(
```

(suite sur la page suivante)

(suite de la page précédente)

```
                    test=Compare(
                        left=Name(id='a', ctx=Load()),
                        ops=[
                            Gt()],
                        comparators=[
                            Constant(value=5)]),
                    body=[
                        Break()],
                    orelse=[
                        Continue()])])])
```

class ast.**Try**(*body*, *handlers*, *orelse*, *finalbody*)

Bloc `try`. Les nœuds de la liste *body* sont les instructions à exécuter sous contrôle des exceptions. La liste *handlers* contient des nœuds `ExceptHandler`, un par `except`. Les listes *orelse* et *finalbody* contiennent respectivement les instructions se trouvant dans d'éventuels blocs `else` et `finally`.

```
>>> print(ast.dump(ast.parse("""
... try:
...     ...
... except Exception:
...     ...
... except OtherException as e:
...     ...
... else:
...     ...
... finally:
...     ...
... """), indent=4))
Module(
    body=[
        Try(
            body=[
                Expr(
                    value=Constant(value=Ellipsis))],
            handlers=[
                ExceptHandler(
                    type=Name(id='Exception', ctx=Load()),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))]),
                ExceptHandler(
                    type=Name(id='OtherException', ctx=Load()),
                    name='e',
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))])],
            orelse=[
                Expr(
                    value=Constant(value=Ellipsis))],
            finalbody=[
                Expr(
                    value=Constant(value=Ellipsis))])])
```

class ast.**TryStar**(*body*, *handlers*, *orelse*, *finalbody*)

`try` blocks which are followed by `except*` clauses. The attributes are the same as for `Try` but the `ExceptHandler` nodes in `handlers` are interpreted as `except*` blocks rather then `except`.

```
>>> print(ast.dump(ast.parse("""
... try:
...     ...
... except* Exception:
...     ...
... """), indent=4))
Module(
    body=[
        TryStar(
            body=[
                Expr(
                    value=Constant(value=Ellipsis))],
            handlers=[
                ExceptHandler(
                    type=Name(id='Exception', ctx=Load()),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))])])])
```

Ajouté dans la version 3.11.

class ast.**ExceptHandler**(*type*, *name*, *body*)

Une clause `except`. Elle intercepte les exceptions de types donnés par *type*, qui est le plus souvent un nœud *Name* ou bien `None` (cette dernière valeur pour les clauses fourre-tout `except :`). Le nom éventuel de la variable à laquelle affecter l'exception est dans la chaîne de caractères *name* (`None` s'il n'y a pas de `as`). Les instructions sous le `except` sont dans la liste *body*.

```
>>> print(ast.dump(ast.parse("""\
... try:
...     a + 1
... except TypeError:
...     pass
... """), indent=4))
Module(
    body=[
        Try(
            body=[
                Expr(
                    value=BinOp(
                        left=Name(id='a', ctx=Load()),
                        op=Add(),
                        right=Constant(value=1)))],
            handlers=[
                ExceptHandler(
                    type=Name(id='TypeError', ctx=Load()),
                    body=[
                        Pass()])])])
```

class ast.**With**(*items*, *body*, *type_comment*)

Bloc `with`. Les gestionnaires de contexte sont stockés dans *items* comme instances de *withitem*. Les instructions sous le `with` forment la liste *body*.

type_comment

Le champ facultatif *type_comment* contient une annotation de type fournie par un commentaire, et ce sous la forme d'une chaîne de caractères.

class ast.**withitem**(*context_expr*, *optional_vars*)

Gestionnaire de contexte dans un bloc `with`. Le gestionnaire est donné par le nœud *context_expr*, souvent une

instance de `Call`. S'il y a affectation avec `as`, *optional_vars* contient sa cible, qui est un nœud de type `Name`, `Tuple` ou `List`. Sinon, *optional_vars* vaut `None`.

```
>>> print(ast.dump(ast.parse("""\
... with a as b, c as d:
...     something(b, d)
... """), indent=4))
Module(
    body=[
        With(
            items=[
                withitem(
                    context_expr=Name(id='a', ctx=Load()),
                    optional_vars=Name(id='b', ctx=Store())),
                withitem(
                    context_expr=Name(id='c', ctx=Load()),
                    optional_vars=Name(id='d', ctx=Store()))],
            body=[
                Expr(
                    value=Call(
                        func=Name(id='something', ctx=Load()),
                        args=[
                            Name(id='b', ctx=Load()),
                            Name(id='d', ctx=Load())]))])])
```

Pattern matching

class `ast.`**`Match`**(*subject*, *cases*)

A `match` statement. `subject` holds the subject of the match (the object that is being matched against the cases) and `cases` contains an iterable of `match_case` nodes with the different cases.

Ajouté dans la version 3.10.

class `ast.`**`match_case`**(*pattern*, *guard*, *body*)

A single case pattern in a `match` statement. `pattern` contains the match pattern that the subject will be matched against. Note that the `AST` nodes produced for patterns differ from those produced for expressions, even when they share the same syntax.

The `guard` attribute contains an expression that will be evaluated if the pattern matches the subject.

`body` contains a list of nodes to execute if the pattern matches and the result of evaluating the guard expression is true.

```
>>> print(ast.dump(ast.parse("""
... match x:
...     case [x] if x>0:
...         ...
...     case tuple():
...         ...
... """), indent=4))
Module(
    body=[
        Match(
            subject=Name(id='x', ctx=Load()),
            cases=[
                match_case(
                    pattern=MatchSequence(
                        patterns=[
                            MatchAs(name='x')]),
```

(suite sur la page suivante)

(suite de la page précédente)

```
                    guard=Compare(
                        left=Name(id='x', ctx=Load()),
                        ops=[
                            Gt()],
                        comparators=[
                            Constant(value=0)]),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))]),
                match_case(
                    pattern=MatchClass(
                        cls=Name(id='tuple', ctx=Load())),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))])])])
```

Ajouté dans la version 3.10.

class ast.**MatchValue**(*value*)

A match literal or value pattern that compares by equality. `value` is an expression node. Permitted value nodes are restricted as described in the match statement documentation. This pattern succeeds if the match subject is equal to the evaluated value.

```
>>> print(ast.dump(ast.parse("""
... match x:
...     case "Relevant":
...         ...
... """), indent=4))
Module(
    body=[
        Match(
            subject=Name(id='x', ctx=Load()),
            cases=[
                match_case(
                    pattern=MatchValue(
                        value=Constant(value='Relevant')),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))])])])
```

Ajouté dans la version 3.10.

class ast.**MatchSingleton**(*value*)

A match literal pattern that compares by identity. `value` is the singleton to be compared against : `None`, `True`, or `False`. This pattern succeeds if the match subject is the given constant.

```
>>> print(ast.dump(ast.parse("""
... match x:
...     case None:
...         ...
... """), indent=4))
Module(
    body=[
        Match(
            subject=Name(id='x', ctx=Load()),
            cases=[
                match_case(
```

(suite sur la page suivante)

(suite de la page précédente)

```
                    pattern=MatchSingleton(value=None),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))])])])
```

Ajouté dans la version 3.10.

class ast.**MatchSequence**(*patterns*)

A match sequence pattern. `patterns` contains the patterns to be matched against the subject elements if the subject is a sequence. Matches a variable length sequence if one of the subpatterns is a `MatchStar` node, otherwise matches a fixed length sequence.

```
>>> print(ast.dump(ast.parse("""
... match x:
...     case [1, 2]:
...         ...
... """), indent=4))
Module(
    body=[
        Match(
            subject=Name(id='x', ctx=Load()),
            cases=[
                match_case(
                    pattern=MatchSequence(
                        patterns=[
                            MatchValue(
                                value=Constant(value=1)),
                            MatchValue(
                                value=Constant(value=2))]),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))])])])
```

Ajouté dans la version 3.10.

class ast.**MatchStar**(*name*)

Matches the rest of the sequence in a variable length match sequence pattern. If `name` is not `None`, a list containing the remaining sequence elements is bound to that name if the overall sequence pattern is successful.

```
>>> print(ast.dump(ast.parse("""
... match x:
...     case [1, 2, *rest]:
...         ...
...     case [*_]:
...         ...
... """), indent=4))
Module(
    body=[
        Match(
            subject=Name(id='x', ctx=Load()),
            cases=[
                match_case(
                    pattern=MatchSequence(
                        patterns=[
                            MatchValue(
                                value=Constant(value=1)),
                            MatchValue(
```

(suite sur la page suivante)

(suite de la page précédente)

```
                                value=Constant(value=2)),
                            MatchStar(name='rest')]),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))]),
                match_case(
                    pattern=MatchSequence(
                        patterns=[
                            MatchStar()]),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))])])])
```

Ajouté dans la version 3.10.

class ast.**MatchMapping**(*keys*, *patterns*, *rest*)

A match mapping pattern. `keys` is a sequence of expression nodes. `patterns` is a corresponding sequence of pattern nodes. `rest` is an optional name that can be specified to capture the remaining mapping elements. Permitted key expressions are restricted as described in the match statement documentation.

This pattern succeeds if the subject is a mapping, all evaluated key expressions are present in the mapping, and the value corresponding to each key matches the corresponding subpattern. If `rest` is not `None`, a dict containing the remaining mapping elements is bound to that name if the overall mapping pattern is successful.

```
>>> print(ast.dump(ast.parse("""
... match x:
...     case {1: _, 2: _}:
...         ...
...     case {**rest}:
...         ...
... """), indent=4))
Module(
    body=[
        Match(
            subject=Name(id='x', ctx=Load()),
            cases=[
                match_case(
                    pattern=MatchMapping(
                        keys=[
                            Constant(value=1),
                            Constant(value=2)],
                        patterns=[
                            MatchAs(),
                            MatchAs()]),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))]),
                match_case(
                    pattern=MatchMapping(rest='rest'),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))])])])
```

Ajouté dans la version 3.10.

class ast.**MatchClass**(*cls*, *patterns*, *kwd_attrs*, *kwd_patterns*)

A match class pattern. `cls` is an expression giving the nominal class to be matched. `patterns` is a sequence of pattern nodes to be matched against the class defined sequence of pattern matching attributes. `kwd_attrs`

is a sequence of additional attributes to be matched (specified as keyword arguments in the class pattern), `kwd_patterns` are the corresponding patterns (specified as keyword values in the class pattern).

This pattern succeeds if the subject is an instance of the nominated class, all positional patterns match the corresponding class-defined attributes, and any specified keyword attributes match their corresponding pattern.

Note : classes may define a property that returns self in order to match a pattern node against the instance being matched. Several builtin types are also matched that way, as described in the match statement documentation.

```
>>> print(ast.dump(ast.parse("""
... match x:
...     case Point2D(0, 0):
...         ...
...     case Point3D(x=0, y=0, z=0):
...         ...
... """), indent=4))
Module(
    body=[
        Match(
            subject=Name(id='x', ctx=Load()),
            cases=[
                match_case(
                    pattern=MatchClass(
                        cls=Name(id='Point2D', ctx=Load()),
                        patterns=[
                            MatchValue(
                                value=Constant(value=0)),
                            MatchValue(
                                value=Constant(value=0))]),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))]),
                match_case(
                    pattern=MatchClass(
                        cls=Name(id='Point3D', ctx=Load()),
                        kwd_attrs=[
                            'x',
                            'y',
                            'z'],
                        kwd_patterns=[
                            MatchValue(
                                value=Constant(value=0)),
                            MatchValue(
                                value=Constant(value=0)),
                            MatchValue(
                                value=Constant(value=0))]),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))])])])
```

Ajouté dans la version 3.10.

class `ast.`**`MatchAs`**(*pattern*, *name*)

A match "as-pattern", capture pattern or wildcard pattern. `pattern` contains the match pattern that the subject will be matched against. If the pattern is `None`, the node represents a capture pattern (i.e a bare name) and will always succeed.

The `name` attribute contains the name that will be bound if the pattern is successful. If `name` is `None`, `pattern` must also be `None` and the node represents the wildcard pattern.

```
>>> print(ast.dump(ast.parse("""
... match x:
...     case [x] as y:
...         ...
...     case _:
...         ...
... """), indent=4))
Module(
    body=[
        Match(
            subject=Name(id='x', ctx=Load()),
            cases=[
                match_case(
                    pattern=MatchAs(
                        pattern=MatchSequence(
                            patterns=[
                                MatchAs(name='x')]),
                        name='y'),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))]),
                match_case(
                    pattern=MatchAs(),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))])])])
```

Ajouté dans la version 3.10.

class ast.**MatchOr**(*patterns*)

A match "or-pattern". An or-pattern matches each of its subpatterns in turn to the subject, until one succeeds. The or-pattern is then deemed to succeed. If none of the subpatterns succeed the or-pattern fails. The `patterns` attribute contains a list of match pattern nodes that will be matched against the subject.

```
>>> print(ast.dump(ast.parse("""
... match x:
...     case [x] | (y):
...         ...
... """), indent=4))
Module(
    body=[
        Match(
            subject=Name(id='x', ctx=Load()),
            cases=[
                match_case(
                    pattern=MatchOr(
                        patterns=[
                            MatchSequence(
                                patterns=[
                                    MatchAs(name='x')]),
                            MatchAs(name='y')]),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))])])])
```

Ajouté dans la version 3.10.

Type parameters

Type parameters can exist on classes, functions, and type aliases.

class ast.**TypeVar**(*name*, *bound*, *default_value*)

A *typing.TypeVar*. name is the name of the type variable. bound is the bound or constraints, if any. If bound is a *Tuple*, it represents constraints ; otherwise it represents the bound. default_value is the default value ; if the TypeVar has no default, this attribute will be set to None.

```
>>> print(ast.dump(ast.parse("type Alias[T: int = bool] = list[T]"), indent=4))
Module(
    body=[
        TypeAlias(
            name=Name(id='Alias', ctx=Store()),
            type_params=[
                TypeVar(
                    name='T',
                    bound=Name(id='int', ctx=Load()),
                    default_value=Name(id='bool', ctx=Load()))],
            value=Subscript(
                value=Name(id='list', ctx=Load()),
                slice=Name(id='T', ctx=Load()),
                ctx=Load()))])
```

Ajouté dans la version 3.12.

Modifié dans la version 3.13 : Added the *default_value* parameter.

class ast.**ParamSpec**(*name*, *default_value*)

A *typing.ParamSpec*. name is the name of the parameter specification. default_value is the default value ; if the ParamSpec has no default, this attribute will be set to None.

```
>>> print(ast.dump(ast.parse("type Alias[**P = [int, str]] = Callable[P, int]
↪"), indent=4))
Module(
    body=[
        TypeAlias(
            name=Name(id='Alias', ctx=Store()),
            type_params=[
                ParamSpec(
                    name='P',
                    default_value=List(
                        elts=[
                            Name(id='int', ctx=Load()),
                            Name(id='str', ctx=Load())],
                        ctx=Load()))],
            value=Subscript(
                value=Name(id='Callable', ctx=Load()),
                slice=Tuple(
                    elts=[
                        Name(id='P', ctx=Load()),
                        Name(id='int', ctx=Load())],
                    ctx=Load()),
                ctx=Load()))])
```

Ajouté dans la version 3.12.

Modifié dans la version 3.13 : Added the *default_value* parameter.

class ast.**TypeVarTuple**(*name*, *default_value*)

A *typing.TypeVarTuple*. name is the name of the type variable tuple. default_value is the default

value ; if the `TypeVarTuple` has no default, this attribute will be set to `None`.

```
>>> print(ast.dump(ast.parse("type Alias[*Ts = ()] = tuple[*Ts]"), indent=4))
Module(
    body=[
        TypeAlias(
            name=Name(id='Alias', ctx=Store()),
            type_params=[
                TypeVarTuple(
                    name='Ts',
                    default_value=Tuple(ctx=Load()))],
            value=Subscript(
                value=Name(id='tuple', ctx=Load()),
                slice=Tuple(
                    elts=[
                        Starred(
                            value=Name(id='Ts', ctx=Load()),
                            ctx=Load())],
                    ctx=Load()),
                ctx=Load()))])
```

Ajouté dans la version 3.12.

Modifié dans la version 3.13 : Added the *default_value* parameter.

Définition de fonctions et de classes

class `ast.`**`FunctionDef`**(*name*, *args*, *body*, *decorator_list*, *returns*, *type_comment*, *type_params*)

Définition d'une fonction.

— *name* donne le nom de la fonction sous forme d'une chaîne de caractères.
— `args` is an `arguments` node.
— *body* est la liste des instructions dans le corps de la fonction.
— *decorators_list* est une liste de décorateurs appliqués à la fonction. Ils sont rangés dans leur ordre d'apparition, c'est-à-dire que le premier de la liste est appliqué en dernier.
— *returns* est l'annotation de renvoi éventuelle.
— `type_params` is a list of *type parameters*.

`type_comment`

Le champ facultatif *type_comment* contient une annotation de type fournie par un commentaire, et ce sous la forme d'une chaîne de caractères.

Modifié dans la version 3.12 : Added `type_params`.

class `ast.`**`Lambda`**(*args*, *body*)

Fonction anonyme. Elle est créée avec le mot-clé `lambda`, limitée à renvoyer une simple expression, et peut être intégrée à une expression plus large. Contrairement à `FunctionDef`, *body* est ici un nœud unique. *args* est une instance de `arguments` qui contient les arguments de la signature.

```
>>> print(ast.dump(ast.parse('lambda x,y: ...'), indent=4))
Module(
    body=[
        Expr(
            value=Lambda(
                args=arguments(
                    args=[
                        arg(arg='x'),
                        arg(arg='y')]),
                body=Constant(value=Ellipsis)))])
```

class `ast.`**`arguments`**(*posonlyargs*, *args*, *vararg*, *kwonlyargs*, *kw_defaults*, *kwarg*, *defaults*)

Arguments d'une fonction.

— *posonlyargs*, *args* et *kwonlyargs* sont des listes de nœuds `arg` qui correspondent respectivement aux arguments obligatoirement positionnels, positionnels ou nommés et obligatoirement nommés.
— *varargs* et *kwargs* sont des nœuds `arg` qui apparaissent avec la capture des arguments restants, positionnels (`*arguments_positionnels`) et nommés (`**arguments_nommés`).
— *kw_defaults* est la liste des valeurs par défaut pour les arguments obligatoirement nommés. Si l'un des éléments n'est pas un nœud mais `None`, il n'y a pas de valeur par défaut et l'argument correspondant doit être passé obligatoirement à la fonction.
— *defaults* est, quant à elle, la liste des valeurs par défauts pour les arguments qui peuvent être passés de manière positionnelle (obligatoirement ou non). S'il y a moins d'éléments dans la liste que d'arguments positionnels, ces éléments donnent les valeurs par défaut des *n* derniers arguments positionnels (avec *n* la longueur de *defaults*).

class `ast.`**`arg`**(*arg*, *annotation*, *type_comment*)

A single argument in a list. `arg` is a raw string of the argument name ; `annotation` is its annotation, such as a `Name` node.

`type_comment`

Le champ facultatif *type_comment* est, sous forme de chaîne, une annotation de type fournie par un commentaire.

```
>>> print(ast.dump(ast.parse("""\
... @decorator1
... @decorator2
... def f(a: 'annotation', b=1, c=2, *d, e, f=3, **g) -> 'return annotation':
...     pass
... """), indent=4))
Module(
    body=[
        FunctionDef(
            name='f',
            args=arguments(
                args=[
                    arg(
                        arg='a',
                        annotation=Constant(value='annotation')),
                    arg(arg='b'),
                    arg(arg='c')],
                vararg=arg(arg='d'),
                kwonlyargs=[
                    arg(arg='e'),
                    arg(arg='f')],
                kw_defaults=[
                    None,
                    Constant(value=3)],
                kwarg=arg(arg='g'),
                defaults=[
                    Constant(value=1),
                    Constant(value=2)]),
            body=[
                Pass()],
            decorator_list=[
                Name(id='decorator1', ctx=Load()),
                Name(id='decorator2', ctx=Load())],
            returns=Constant(value='return annotation'))])
```

class `ast.`**`Return`**(*value*)

Instruction `return`, qui renvoie *value*.

```
>>> print(ast.dump(ast.parse('return 4'), indent=4))
Module(
    body=[
        Return(
            value=Constant(value=4))])
```

class ast.**Yield**(*value*)

class ast.**YieldFrom**(*value*)

A `yield` or `yield from` expression. Because these are expressions, they must be wrapped in an *Expr* node if the value sent back is not used.

```
>>> print(ast.dump(ast.parse('yield x'), indent=4))
Module(
    body=[
        Expr(
            value=Yield(
                value=Name(id='x', ctx=Load())))])

>>> print(ast.dump(ast.parse('yield from x'), indent=4))
Module(
    body=[
        Expr(
            value=YieldFrom(
                value=Name(id='x', ctx=Load())))])
```

class ast.**Global**(*names*)

class ast.**Nonlocal**(*names*)

Instruction `global` ou `nonlocal`. Elle s'applique aux noms de variables donnés dans *names*, une liste de chaînes de caractères.

```
>>> print(ast.dump(ast.parse('global x,y,z'), indent=4))
Module(
    body=[
        Global(
            names=[
                'x',
                'y',
                'z'])])

>>> print(ast.dump(ast.parse('nonlocal x,y,z'), indent=4))
Module(
    body=[
        Nonlocal(
            names=[
                'x',
                'y',
                'z'])])
```

class ast.**ClassDef**(*name*, *bases*, *keywords*, *body*, *decorator_list*, *type_params*)

Définition d'une classe.

— *name* est le nom de la classe sous forme de chaîne de caractères ;
— *bases* est la liste des classes mères données explicitement ;
— `keywords` is a list of *keyword* nodes, principally for 'metaclass'. Other keywords will be passed to the metaclass, as per **PEP 3115**.
— *body* est la liste des instructions contenues dans la définition de classe ;
— *decorators_list* est une liste de nœuds, comme pour *FunctionDef*.
— `type_params` is a list of *type parameters*.

```
>>> print(ast.dump(ast.parse("""\
... @decorator1
... @decorator2
... class Foo(base1, base2, metaclass=meta):
...     pass
... """), indent=4))
Module(
    body=[
        ClassDef(
            name='Foo',
            bases=[
                Name(id='base1', ctx=Load()),
                Name(id='base2', ctx=Load())],
            keywords=[
                keyword(
                    arg='metaclass',
                    value=Name(id='meta', ctx=Load()))],
            body=[
                Pass()],
            decorator_list=[
                Name(id='decorator1', ctx=Load()),
                Name(id='decorator2', ctx=Load())])])
```

Modifié dans la version 3.12 : Added `type_params`.

`async` et `await`

class ast.**AsyncFunctionDef**(*name*, *args*, *body*, *decorator_list*, *returns*, *type_comment*, *type_params*)

Fonction déclarée avec `async def`. Les champs sont les mêmes que dans *FunctionDef*.

Modifié dans la version 3.12 : Added `type_params`.

class ast.**Await**(*value*)

Expression `await`, qui attend *value*. Ces nœuds ne peuvent apparaître qu'à l'intérieur de *AsyncFunctionDef*.

```
>>> print(ast.dump(ast.parse("""\
... async def f():
...     await other_func()
... """), indent=4))
Module(
    body=[
        AsyncFunctionDef(
            name='f',
            args=arguments(),
            body=[
                Expr(
                    value=Await(
                        value=Call(
                            func=Name(id='other_func', ctx=Load())))])])
```

class ast.**AsyncFor**(*target*, *iter*, *body*, *orelse*, *type_comment*)

class ast.**AsyncWith**(*items*, *body*, *type_comment*)

Instruction `async for` ou `async with`. Les champs sont les mêmes que ceux de *For* et *With*. Ces nœuds ne peuvent apparaître qu'à l'intérieur de *AsyncFunctionDef*.

Note

Lorsqu'une chaîne contenant du code est transformée en arbre syntaxique par `ast.parse()`, les nœuds représentants les opérateurs (classes filles de `ast.operator`, `ast.unaryop`, `ast.cmpop`, `ast.boolop` et `ast.expr_context`) sont des singletons à l'intérieur de l'arbre renvoyé. Si l'un, par exemple une instance de `ast.Add`, est muté, la modification sera visible sur toutes les autres apparitions de l'opérateur.

33.1.3 Outils du module `ast`

À part les classes de nœuds, le module `ast` définit plusieurs fonctions et classes utilitaires pour traverser les arbres syntaxiques abstraits :

`ast.`**`parse`**(*source*, *filename='<unknown>'*, *mode='exec'*, **, type_comments=False*, *feature_version=None*, *optimize=-1*)

Parse the source into an AST node. Equivalent to `compile(source, filename, mode, flags=FLAGS_VALUE, optimize=optimize)`, where `FLAGS_VALUE` is `ast.PyCF_ONLY_AST` if `optimize <= 0` and `ast.PyCF_OPTIMIZED_AST` otherwise.

If `type_comments=True` is given, the parser is modified to check and return type comments as specified by **PEP 484** and **PEP 526**. This is equivalent to adding `ast.PyCF_TYPE_COMMENTS` to the flags passed to `compile()`. This will report syntax errors for misplaced type comments. Without this flag, type comments will be ignored, and the `type_comment` field on selected AST nodes will always be `None`. In addition, the locations of `# type: ignore` comments will be returned as the `type_ignores` attribute of `Module` (otherwise it is always an empty list).

In addition, if `mode` is `'func_type'`, the input syntax is modified to correspond to **PEP 484** "signature type comments", e.g. `(str, int) -> List[str]`.

Setting `feature_version` to a tuple `(major, minor)` will result in a "best-effort" attempt to parse using that Python version's grammar. For example, setting `feature_version=(3, 9)` will attempt to disallow parsing of `match` statements. Currently `major` must equal to `3`. The lowest supported version is `(3, 7)` (and this may increase in future Python versions) ; the highest is `sys.version_info[0:2]`. "Best-effort" attempt means there is no guarantee that the parse (or success of the parse) is the same as when run on the Python version corresponding to `feature_version`.

If source contains a null character (`\0`), `ValueError` is raised.

> **Avertissement**
>
> Note that successfully parsing source code into an AST object doesn't guarantee that the source code provided is valid Python code that can be executed as the compilation step can raise further `SyntaxError` exceptions. For instance, the source `return 42` generates a valid AST node for a return statement, but it cannot be compiled alone (it needs to be inside a function node).
>
> In particular, `ast.parse()` won't do any scoping checks, which the compilation step does.

> **Avertissement**
>
> Il est possible de faire planter l'interpréteur Python avec des chaînes suffisamment grandes ou complexes lors de la compilation d'un arbre syntaxique en raison de la limitation de la profondeur de la pile d'appels.

Modifié dans la version 3.8 : ajout des paramètres *type_comments* et *feature_version* ainsi que de la valeur `'func_type'` pour *mode*.

Modifié dans la version 3.13 : The minimum supported version for `feature_version` is now `(3, 7)`. The `optimize` argument was added.

`ast.`**`unparse`**(*ast_obj*)

À partir d'un arbre syntaxique `ast.AST`, reconstruit un code sous forme de chaîne de caractères. S'il est passé

à `ast.parse()`, le résultat produit un arbre `ast.AST` équivalent à l'original.

> **Avertissement**
>
> The produced code string will not necessarily be equal to the original code that generated the `ast.AST` object (without any compiler optimizations, such as constant tuples/frozensets).

> **Avertissement**
>
> Une `RecursionError` est levée si l'expression comporte de très nombreux niveaux d'imbrication.

Ajouté dans la version 3.9.

`ast.literal_eval(node_or_string)`

Evaluate an expression node or a string containing only a Python literal or container display. The string or node provided may only consist of the following Python literal structures : strings, bytes, numbers, tuples, lists, dicts, sets, booleans, `None` and `Ellipsis`.

This can be used for evaluating strings containing Python values without the need to parse the values oneself. It is not capable of evaluating arbitrarily complex expressions, for example involving operators or indexing.

This function had been documented as "safe" in the past without defining what that meant. That was misleading. This is specifically designed not to execute Python code, unlike the more general `eval()`. There is no namespace, no name lookups, or ability to call out. But it is not free from attack : A relatively small input can lead to memory exhaustion or to C stack exhaustion, crashing the process. There is also the possibility for excessive CPU consumption denial of service on some inputs. Calling it on untrusted data is thus not recommended.

> **Avertissement**
>
> It is possible to crash the Python interpreter due to stack depth limitations in Python's AST compiler.
>
> It can raise `ValueError`, `TypeError`, `SyntaxError`, `MemoryError` and `RecursionError` depending on the malformed input.

Modifié dans la version 3.2 : accepte maintenant les octets et ensembles littéraux.

Modifié dans la version 3.9 : accepte `set()` pour les ensembles vides.

Modifié dans la version 3.10 : For string inputs, leading spaces and tabs are now stripped.

`ast.get_docstring(node, clean=True)`

Renvoie la *docstring* du nœud *node* (de type `FunctionDef`, `AsyncFunctionDef`, `ClassDef` or `Module`), ou `None` s'il n'a pas de *docstring*. Si *clean* est vrai, cette fonction nettoie l'indentation de la *docstring* avec `inspect.cleandoc()`.

Modifié dans la version 3.5 : `AsyncFunctionDef` est maintenant gérée.

`ast.get_source_segment(source, node, *, padded=False)`

Get source code segment of the *source* that generated *node*. If some location information (`lineno`, `end_lineno`, `col_offset`, or `end_col_offset`) is missing, return `None`.

If *padded* is `True`, the first line of a multi-line statement will be padded with spaces to match its original position.

Ajouté dans la version 3.8.

`ast.fix_missing_locations(node)`

When you compile a node tree with `compile()`, the compiler expects `lineno` and `col_offset` attributes for every node that supports them. This is rather tedious to fill in for generated nodes, so this helper adds

these attributes recursively where not already set, by setting them to the values of the parent node. It works recursively starting at *node*.

`ast.increment_lineno(node, n=1)`

Incrémente de *n* les numéros des lignes de début et ligne de fin de chaque nœud dans l'arbre, en commençant par le nœud *node*. C'est utile pour « déplacer du code » à un endroit différent dans un fichier.

`ast.copy_location(new_node, old_node)`

Copy source location (`lineno`, `col_offset`, `end_lineno`, and `end_col_offset`) from *old_node* to *new_node* if possible, and return *new_node*.

`ast.iter_fields(node)`

Produit un *n*-uplet de couples `(nom_du_champ, valeur)` pour chaque champ de `node._fields` qui est présent dans *node*.

`ast.iter_child_nodes(node)`

Produit tous les nœuds enfants directs de *node*, c'est-à-dire tous les champs qui sont des nœuds et tous les éléments des champs qui sont des listes de nœuds.

`ast.walk(node)`

Produit récursivement tous les nœuds enfants dans l'arbre en commençant par *node* (*node* lui-même est inclus), sans ordre spécifique. C'est utile lorsque l'on souhaite modifier les nœuds sur place sans prêter attention au contexte.

class `ast.NodeVisitor`

Classe de base pour un visiteur de nœud, qui parcourt l'arbre syntaxique abstrait et appelle une fonction de visite pour chacun des nœuds trouvés. Cette fonction peut renvoyer une valeur, qui est transmise par la méthode `visit()`.

Cette classe est faite pour être dérivée, en ajoutant des méthodes de visite à la sous-classe.

visit(*node*)

Visite un nœud. L'implémentation par défaut appelle la méthode `self.visit_classe` où *classe* représente le nom de la classe du nœud, ou `generic_visit()` si cette méthode n'existe pas.

generic_visit(*node*)

Le visiteur appelle la méthode `visit()` de tous les enfants du nœud.

Notons que les nœuds enfants qui possèdent une méthode de visite spéciale ne sont pas visités à moins que le visiteur n'appelle la méthode `generic_visit()` ou ne les visite lui-même.

visit_Constant(*node*)

Handles all constant nodes.

N'utilisez pas `NodeVisitor` si vous souhaitez appliquer des changements sur les nœuds lors du parcours. Pour cela, un visiteur spécial existe (`NodeTransformer`) qui permet les modifications.

Obsolète depuis la version 3.8 : Methods `visit_Num()`, `visit_Str()`, `visit_Bytes()`, `visit_NameConstant()` and `visit_Ellipsis()` are deprecated now and will not be called in future Python versions. Add the `visit_Constant()` method to handle all constant nodes.

class `ast.NodeTransformer`

Une sous-classe de `NodeVisitor` qui traverse l'arbre syntaxique abstrait et permet de modifier les nœuds.

Le `NodeTransformer` traverse l'arbre syntaxique et utilise la valeur renvoyée par les méthodes du visiteur pour remplacer ou supprimer l'ancien nœud. Si la valeur renvoyée par la méthode du visiteur est `None`, le nœud est supprimé de sa position, sinon il est remplacé par cette valeur. Elle peut être le nœud original, auquel cas il n'y a pas de remplacement.

Voici un exemple de transformation qui réécrit tous les accès à la valeur d'une variable `toto` en `data['toto']` :

```
class RewriteName(NodeTransformer):

    def visit_Name(self, node):
        return Subscript(
            value=Name(id='data', ctx=Load()),
            slice=Constant(value=node.id),
            ctx=node.ctx
        )
```

Keep in mind that if the node you're operating on has child nodes you must either transform the child nodes yourself or call the *generic_visit()* method for the node first.

Pour les nœuds qui font partie d'un groupe (notamment toutes les instructions), le visiteur peut aussi renvoyer une liste des nœuds plutôt qu'un seul nœud.

If *NodeTransformer* introduces new nodes (that weren't part of original tree) without giving them location information (such as *lineno*), *fix_missing_locations()* should be called with the new sub-tree to recalculate the location information :

```
tree = ast.parse('foo', mode='eval')
new_tree = fix_missing_locations(RewriteName().visit(tree))
```

Utilisation typique des transformations :

```
node = YourTransformer().visit(node)
```

`ast.`**`dump`**(*node*, *annotate_fields=True*, *include_attributes=False*, ***, *indent=None*, *show_empty=False*)

Renvoie une représentation sous forme de chaîne de caractères de l'arbre contenu dans *node*. Ceci est principalement utile à des fins de débogage. Par défaut, les champs sont passés comme paramètres nommés aux constructeurs des classes d'arbre syntaxiques. Cependant, si *annotate_fields* est mis à `False`, les valeurs sont passées, lorsque cela est possible, comme arguments positionnels, rendant la représentation plus compacte. Les attributs comme les numéros de lignes et positions sur les lignes sont masqués par défaut, mais on peut les inclure en mettant *include_attributes* à `True`.

La représentation peut comprendre une indentation afin d'être plus lisible. Si *indent* est une chaîne de caractères (par exemple une tabulation `"\t"`), elle est insérée au début des lignes en la répétant autant de fois que le niveau d'indentation. Un entier positif équivaut à un certain nombre d'espaces. Un entier strictement négatif produit un effet identique à 0 ou la chaîne vide, c'est-à-dire des retours à la ligne sans indentation. Avec la valeur par défaut de `None`, la sortie tient sur une seule ligne.

If *show_empty* is `False` (the default), empty lists and fields that are `None` will be omitted from the output.

Modifié dans la version 3.9 : ajout du paramètre *indent*.

Modifié dans la version 3.13 : Added the *show_empty* option.

```
>>> print(ast.dump(ast.parse("""\
... async def f():
...     await other_func()
... """), indent=4, show_empty=True))
Module(
    body=[
        AsyncFunctionDef(
            name='f',
            args=arguments(
                posonlyargs=[],
                args=[],
                kwonlyargs=[],
                kw_defaults=[],
                defaults=[]),
```

(suite sur la page suivante)

(suite de la page précédente)

```
            body=[
                Expr(
                    value=Await(
                        value=Call(
                            func=Name(id='other_func', ctx=Load()),
                            args=[],
                            keywords=[]))),
            decorator_list=[],
            type_params=[])],
    type_ignores=[])
```

33.1.4 Options du compilateur

Les options suivantes sont prises en charge par la fonction `compile()`. Elles permettent de modifier le comportement de la compilation :

`ast.`**`PyCF_ALLOW_TOP_LEVEL_AWAIT`**

Active la reconnaissance de `await`, `async for`, `async with` et des compréhensions asynchrones au niveau le plus haut.

Ajouté dans la version 3.8.

`ast.`**`PyCF_ONLY_AST`**

Génère et renvoie un arbre syntaxique au lieu d'un objet de code compilé.

`ast.`**`PyCF_OPTIMIZED_AST`**

The returned AST is optimized according to the *optimize* argument in `compile()` or `ast.parse()`.

Ajouté dans la version 3.13.

`ast.`**`PyCF_TYPE_COMMENTS`**

Ajoute la prise en charge des commentaires de types tels que définis dans la **PEP 484** et la **PEP 526** (`# type: un_type` et `# type: ignore`).

Ajouté dans la version 3.8.

`ast.`**`compare`**(*a*, *b*, /, *, *compare_attributes=False*)

Recursively compares two ASTs.

compare_attributes affects whether AST attributes are considered in the comparison. If *compare_attributes* is `False` (default), then attributes are ignored. Otherwise they must all be equal. This option is useful to check whether the ASTs are structurally equal but differ in whitespace or similar details. Attributes include line numbers and column offsets.

Ajouté dans la version 3.14.

33.1.5 Utilisation en ligne de commande

Ajouté dans la version 3.9.

Le module `ast` peut être exécuté en tant que script en ligne de commande. C'est aussi simple que ceci :

```
python -m ast [-m <mode>] [-a] [infile]
```

Les options suivantes sont acceptées :

`-h, --help`

Affiche un message d'aide et quitte.

`-m` `<mode>`

--mode <mode>

Précise le type de code à compiler, comme l'argument *mode* de la fonction `parse()`.

--no-type-comments

Désactive la reconnaissance des commentaires de type.

-a, --include-attributes

Affiche les attributs comme les numéros de lignes et les décalages par rapport aux débuts des lignes.

-i <indent>

--indent <indent>

Nombre d'espaces pour chaque niveau d'indentation dans la sortie.

L'entrée est lue dans le fichier `infile`, s'il est donné, ou l'entrée standard sinon. Le code source est transformé en un arbre syntaxique, qui est affiché sur la sortie standard.

Voir aussi

Green Tree Snakes, une ressource documentaire externe, qui possède plus de détails pour travailler avec des arbres syntaxiques Python.

ASTTokens annote les arbres syntaxiques Python avec les positions des lexèmes et les extraits de code source à partir desquels ils sont produits. Ceci est utile pour les outils qui transforment du code source.

leoAst.py unifies the token-based and parse-tree-based views of python programs by inserting two-way links between tokens and ast nodes.

LibCST produit à partir du code source des arbres syntaxiques concrets, qui ressemblent à leurs homologues abstraits et conservent tous les détails du formatage. Cette bibliothèque est utile aux outils de réusinage et d'analyse de code.

Parso is a Python parser that supports error recovery and round-trip parsing for different Python versions (in multiple Python versions). Parso is also able to list multiple syntax errors in your Python file.

33.2 `symtable` --- Access to the compiler's symbol tables

Code source : Lib/symtable.py

Symbol tables are generated by the compiler from AST just before bytecode is generated. The symbol table is responsible for calculating the scope of every identifier in the code. `symtable` provides an interface to examine these tables.

33.2.1 Generating Symbol Tables

symtable.**symtable**(*code*, *filename*, *compile_type*)

Return the toplevel `SymbolTable` for the Python source *code*. *filename* is the name of the file containing the code. *compile_type* is like the *mode* argument to `compile()`.

33.2.2 Examining Symbol Tables

class symtable.**SymbolTableType**

An enumeration indicating the type of a `SymbolTable` object.

MODULE = "module"

Used for the symbol table of a module.

FUNCTION = "function"

Used for the symbol table of a function.

CLASS = "class"

Used for the symbol table of a class.

The following members refer to different flavors of annotation scopes.

ANNOTATION = "annotation"

Used for annotations if `from __future__ import annotations` is active.

TYPE_ALIAS = "type alias"

Used for the symbol table of `type` constructions.

TYPE_PARAMETERS = "type parameters"

Used for the symbol table of generic functions or generic classes.

TYPE_VARIABLE = "type variable"

Used for the symbol table of the bound, the constraint tuple or the default value of a single type variable in the formal sense, i.e., a TypeVar, a TypeVarTuple or a ParamSpec object (the latter two do not support a bound or a constraint tuple).

Ajouté dans la version 3.13.

class symtable.**SymbolTable**

A namespace table for a block. The constructor is not public.

get_type()

Return the type of the symbol table. Possible values are members of the *SymbolTableType* enumeration.

Modifié dans la version 3.12 : Added `'annotation'`, `'TypeVar bound'`, `'type alias'`, and `'type parameter'` as possible return values.

Modifié dans la version 3.13 : Return values are members of the *SymbolTableType* enumeration.

The exact values of the returned string may change in the future, and thus, it is recommended to use *SymbolTableType* members instead of hard-coded strings.

get_id()

Return the table's identifier.

get_name()

Return the table's name. This is the name of the class if the table is for a class, the name of the function if the table is for a function, or `'top'` if the table is global (*get_type()* returns `'module'`). For type parameter scopes (which are used for generic classes, functions, and type aliases), it is the name of the underlying class, function, or type alias. For type alias scopes, it is the name of the type alias. For *TypeVar* bound scopes, it is the name of the `TypeVar`.

get_lineno()

Return the number of the first line in the block this table represents.

is_optimized()

Return `True` if the locals in this table can be optimized.

is_nested()

Return `True` if the block is a nested class or function.

has_children()

Return `True` if the block has nested namespaces within it. These can be obtained with *get_children()*.

get_identifiers()

Return a view object containing the names of symbols in the table. See the *documentation of view objects*.

lookup(*name*)

Lookup *name* in the table and return a *Symbol* instance.

get_symbols()

Return a list of *Symbol* instances for names in the table.

get_children()

Return a list of the nested symbol tables.

class symtable.**Function**

A namespace for a function or method. This class inherits from *SymbolTable*.

get_parameters()

Return a tuple containing names of parameters to this function.

get_locals()

Return a tuple containing names of locals in this function.

get_globals()

Return a tuple containing names of globals in this function.

get_nonlocals()

Return a tuple containing names of explicitly declared nonlocals in this function.

get_frees()

Return a tuple containing names of *free (closure) variables* in this function.

class symtable.**Class**

A namespace of a class. This class inherits from *SymbolTable*.

get_methods()

Return a tuple containing the names of method-like functions declared in the class.

Here, the term 'method' designates *any* function defined in the class body via `def` or `async def`.

Functions defined in a deeper scope (e.g., in an inner class) are not picked up by *get_methods()*.

For example :

```
>>> import symtable
>>> st = symtable.symtable('''
... def outer(): pass
...
... class A:
...     def f():
...         def w(): pass
...
...     def g(self): pass
...
...     @classmethod
...     async def h(cls): pass
...
...     global outer
...     def outer(self): pass
... ''', 'test', 'exec')
>>> class_A = st.get_children()[2]
>>> class_A.get_methods()
('f', 'g', 'h')
```

Although `A().f()` raises *TypeError* at runtime, `A.f` is still considered as a method-like function.

Deprecated since version 3.14, will be removed in version 3.16.

class symtable.**Symbol**

An entry in a *SymbolTable* corresponding to an identifier in the source. The constructor is not public.

get_name()

Return the symbol's name.

is_referenced()

Return `True` if the symbol is used in its block.

is_imported()

Return `True` if the symbol is created from an import statement.

is_parameter()

Return `True` if the symbol is a parameter.

is_type_parameter()

Return `True` if the symbol is a type parameter.

Ajouté dans la version 3.14.

is_global()

Return `True` if the symbol is global.

is_nonlocal()

Return `True` if the symbol is nonlocal.

is_declared_global()

Return `True` if the symbol is declared global with a global statement.

is_local()

Return `True` if the symbol is local to its block.

is_annotated()

Return `True` if the symbol is annotated.

Ajouté dans la version 3.6.

is_free()

Return `True` if the symbol is referenced in its block, but not assigned to.

is_free_class()

Return *True* if a class-scoped symbol is free from the perspective of a method.

Consider the following example :

```
def f():
    x = 1  # function-scoped
    class C:
        x = 2  # class-scoped
        def method(self):
            return x
```

In this example, the class-scoped symbol `x` is considered to be free from the perspective of `C.method`, thereby allowing the latter to return *1* at runtime and not *2*.

Ajouté dans la version 3.14.

is_assigned()

Return `True` if the symbol is assigned to in its block.

is_comp_iter()

Return `True` if the symbol is a comprehension iteration variable.

Ajouté dans la version 3.14.

is_comp_cell()

Return `True` if the symbol is a cell in an inlined comprehension.

Ajouté dans la version 3.14.

is_namespace()

Return `True` if name binding introduces new namespace.

If the name is used as the target of a function or class statement, this will be true.

Par exemple :

```
>>> table = symtable.symtable("def some_func(): pass", "string", "exec")
>>> table.lookup("some_func").is_namespace()
True
```

Note that a single name can be bound to multiple objects. If the result is `True`, the name may also be bound to other objects, like an int or list, that does not introduce a new namespace.

get_namespaces()

Return a list of namespaces bound to this name.

get_namespace()

Return the namespace bound to this name. If more than one or no namespace is bound to this name, a `ValueError` is raised.

33.2.3 Utilisation en ligne de commande.

Ajouté dans la version 3.13.

The `symtable` module can be executed as a script from the command line.

```
python -m symtable [infile...]
```

Symbol tables are generated for the specified Python source files and dumped to stdout. If no input file is specified, the content is read from stdin.

33.3 `token` --- Constants used with Python parse trees

Code source : Lib/token.py

This module provides constants which represent the numeric values of leaf nodes of the parse tree (terminal tokens). Refer to the file `Grammar/Tokens` in the Python distribution for the definitions of the names in the context of the language grammar. The specific numeric values which the names map to may change between Python versions.

Le module fournit également une correspondance entre les codes numériques et les noms et certaines fonctions. Les fonctions reflètent les définitions des fichiers d'en-tête C de Python.

`token.`**tok_name**

Dictionnaire faisant correspondre les valeurs numériques des constantes définies dans ce module à leurs noms, permettant de générer une représentation plus humaine des arbres syntaxiques.

`token.`**ISTERMINAL**(*x*)

Return `True` for terminal token values.

`token.`**ISNONTERMINAL**(*x*)

Return `True` for non-terminal token values.

`token.`**ISEOF**(*x*)

Return `True` if *x* is the marker indicating the end of input.

Les constantes associées aux jetons sont :

`token.`**ENDMARKER**

token.**NAME**

token.**NUMBER**

token.**STRING**

token.**NEWLINE**

token.**INDENT**

token.**DEDENT**

token.**LPAR**
Token value for "(".

token.**RPAR**
Token value for ")".

token.**LSQB**
Token value for "[".

token.**RSQB**
Token value for "]".

token.**COLON**
Token value for ":".

token.**COMMA**
Token value for ",".

token.**SEMI**
Token value for ";".

token.**PLUS**
Token value for "+".

token.**MINUS**
Token value for "-".

token.**STAR**
Token value for "*".

token.**SLASH**
Token value for "/".

token.**VBAR**
Token value for "|".

token.**AMPER**
Token value for "&".

token.**LESS**
Token value for "<".

token.**GREATER**
Token value for ">".

token.**EQUAL**
Token value for "=".

token.**DOT**
Token value for ".".

token.**PERCENT**

Token value for "%".

token.**LBRACE**

Token value for "{".

token.**RBRACE**

Token value for "}".

token.**EQEQUAL**

Token value for "==".

token.**NOTEQUAL**

Token value for "!=".

token.**LESSEQUAL**

Token value for "<=".

token.**GREATEREQUAL**

Token value for ">=".

token.**TILDE**

Token value for "~".

token.**CIRCUMFLEX**

Token value for "^".

token.**LEFTSHIFT**

Token value for "<<".

token.**RIGHTSHIFT**

Token value for ">>".

token.**DOUBLESTAR**

Token value for "**".

token.**PLUSEQUAL**

Token value for "+=".

token.**MINEQUAL**

Token value for "-=".

token.**STAREQUAL**

Token value for "*=".

token.**SLASHEQUAL**

Token value for "/=".

token.**PERCENTEQUAL**

Token value for "%=".

token.**AMPEREQUAL**

Token value for "&=".

token.**VBAREQUAL**

Token value for "|=".

token.**CIRCUMFLEXEQUAL**

Token value for "^=".

token.**LEFTSHIFTEQUAL**

Token value for "<<=".

token.**RIGHTSHIFTEQUAL**

Token value for `">>="`.

token.**DOUBLESTAREQUAL**

Token value for `"**="`.

token.**DOUBLESLASH**

Token value for `"//"`.

token.**DOUBLESLASHEQUAL**

Token value for `"//="`.

token.**AT**

Token value for `"@"`.

token.**ATEQUAL**

Token value for `"@="`.

token.**RARROW**

Token value for `"->"`.

token.**ELLIPSIS**

Token value for `"..."`.

token.**COLONEQUAL**

Token value for `":="`.

token.**EXCLAMATION**

Token value for `"!"`.

token.**OP**

token.**TYPE_IGNORE**

token.**TYPE_COMMENT**

token.**SOFT_KEYWORD**

token.**FSTRING_START**

token.**FSTRING_MIDDLE**

token.**FSTRING_END**

token.**COMMENT**

token.**NL**

token.**ERRORTOKEN**

token.**N_TOKENS**

token.**NT_OFFSET**

Les types de jetons suivants ne sont pas utilisés par l'analyseur lexical C mais sont requis par le module *tokenize*.

token.**COMMENT**

Valeur du jeton utilisée pour indiquer un commentaire.

token.**NL**

Valeur du jeton utilisée pour indiquer un retour à la ligne non terminal. Le jeton *NEWLINE* indique la fin d'une ligne logique de code Python ; les jetons `NL` sont générés quand une ligne logique de code s'étend sur plusieurs lignes.

token.**ENCODING**

Valeur de jeton qui indique l'encodage utilisé pour décoder le fichier initial. Le premier jeton renvoyé par *tokenize.tokenize()* sera toujours un jeton ENCODING.

token.**TYPE_COMMENT**

Token value indicating that a type comment was recognized. Such tokens are only produced when *ast.parse()* is invoked with type_comments=True.

token.**EXACT_TOKEN_TYPES**

A dictionary mapping the string representation of a token to its numeric code.

Ajouté dans la version 3.8.

Modifié dans la version 3.5 : Added AWAIT and ASYNC tokens.

Modifié dans la version 3.7 : Ajout des jetons *COMMENT*, *NL* et *ENCODING*.

Modifié dans la version 3.7 : Removed AWAIT and ASYNC tokens. "async" and "await" are now tokenized as *NAME* tokens.

Modifié dans la version 3.8 : Added *TYPE_COMMENT*, *TYPE_IGNORE*, *COLONEQUAL*. Added AWAIT and ASYNC tokens back (they're needed to support parsing older Python versions for *ast.parse()* with feature_version set to 6 or lower).

Modifié dans la version 3.13 : Removed AWAIT and ASYNC tokens again.

33.4 keyword --- Testing for Python keywords

Code source : Lib/keyword.py

This module allows a Python program to determine if a string is a keyword or soft keyword.

keyword.**iskeyword**(*s*)

Renvoie vrai si *s* est un mot-clé de Python.

keyword.**kwlist**

Séquence contenant tous les mots-clés définis par l'interpréteur. Si certains mots-clés sont définis pour être actifs seulement si des instructions *__future__* particulières sont effectives, ils seront tout de même inclus.

keyword.**issoftkeyword**(*s*)

Return True if *s* is a Python soft keyword.

Ajouté dans la version 3.9.

keyword.**softkwlist**

Sequence containing all the soft keywords defined for the interpreter. If any soft keywords are defined to only be active when particular *__future__* statements are in effect, these will be included as well.

Ajouté dans la version 3.9.

33.5 tokenize --- Tokenizer for Python source

Code source : Lib/tokenize.py

The *tokenize* module provides a lexical scanner for Python source code, implemented in Python. The scanner in this module returns comments as tokens as well, making it useful for implementing "pretty-printers", including colorizers for on-screen displays.

Pour simplifier la gestion de flux de *tokens*, tous les *tokens* operator et delimiter, ainsi que les Ellipsis* sont renvoyés en utilisant le *token* générique *OP*. Le type exact peut être déterminé en vérifiant la propriété exact_type du *named tuple* renvoyé par *tokenize.tokenize()*.

Avertissement

Note that the functions in this module are only designed to parse syntactically valid Python code (code that does not raise when parsed using `ast.parse()`). The behavior of the functions in this module is **undefined** when providing invalid Python code and it can change at any point.

33.5.1 Analyse Lexicale

Le point d'entrée principal est un *générateur* :

`tokenize.tokenize`(*readline*)

Le générateur `tokenize()` prend un argument *readline* qui doit être un objet appelable exposant la même interface que la méthode `io.IOBase.readline()` des objets fichiers. Chaque appel a la fonction doit renvoyer une ligne sous forme de *bytes*.

The generator produces 5-tuples with these members : the token type ; the token string ; a 2-tuple `(srow, scol)` of ints specifying the row and column where the token begins in the source ; a 2-tuple `(erow, ecol)` of ints specifying the row and column where the token ends in the source ; and the line on which the token was found. The line passed (the last tuple item) is the *physical* line. The 5 tuple is returned as a *named tuple* with the field names : `type string start end line`.

Le *n-uplet nommé* a une propriété additionnelle appelée `exact_type` qui contient le type exact de l'opérateur pour les jetons `OP`. Pour tous les autres types de jetons, `exact_type` est égal au champ `type` du *n*-uplet nommé.

Modifié dans la version 3.1 : prise en charge des *n*-uplets nommés.

Modifié dans la version 3.3 : prise en charge de `exact_type`.

`tokenize()` détermine le codage source du fichier en recherchant une nomenclature UTF-8 ou un cookie d'encodage, selon la **PEP 263**.

`tokenize.generate_tokens`(*readline*)

Tokenize a source reading unicode strings instead of bytes.

Like `tokenize()`, the *readline* argument is a callable returning a single line of input. However, `generate_tokens()` expects *readline* to return a str object rather than bytes.

The result is an iterator yielding named tuples, exactly like `tokenize()`. It does not yield an `ENCODING` token.

Toutes les constantes du module `token` sont également exportées depuis module `tokenize`.

Une autre fonction est fournie pour inverser le processus de tokenisation. Ceci est utile pour créer des outils permettant de codifier un script, de modifier le flux de jetons et de réécrire le script modifié.

`tokenize.untokenize`(*iterable*)

Convertit les jetons en code source Python. L'*iterable* doit renvoyer des séquences avec au moins deux éléments, le type de jeton et la chaîne de caractères associée. Tout élément de séquence supplémentaire est ignoré.

The result is guaranteed to tokenize back to match the input so that the conversion is lossless and round-trips are assured. The guarantee applies only to the token type and token string as the spacing between tokens (column positions) may change.

It returns bytes, encoded using the `ENCODING` token, which is the first token sequence output by `tokenize()`. If there is no encoding token in the input, it returns a str instead.

`tokenize()` a besoin de détecter le codage des fichiers sources qu'il code. La fonction utilisée pour cela est disponible :

`tokenize.detect_encoding`(*readline*)

La fonction `detect_encoding()` est utilisée pour détecter l'encodage à utiliser pour décoder un fichier source Python. Il nécessite un seul argument, *readline*, de la même manière que le générateur `tokenize()`.

Il appelle *readline* au maximum deux fois et renvoie le codage utilisé (sous forme de chaîne) et une liste de toutes les lignes (non décodées à partir des octets) dans lesquelles il a été lu.

Il détecte l'encodage par la présence d'une marqueur UTF-8 (*BOM*) ou d'un cookie de codage, comme spécifié dans la **PEP 263**. Si un *BOM* et un cookie sont présents, mais en désaccord, un `SyntaxError` sera levée. Notez que si le *BOM* est trouvé, `'utf-8-sig'` sera renvoyé comme encodage.

Si aucun codage n'est spécifié, la valeur par défaut, `'utf-8'`, sera renvoyée.

Utilisez `open()` pour ouvrir les fichiers source Python : ça utilise `detect_encoding()` pour détecter le codage du fichier.

`tokenize.`**`open`**(*filename*)

Ouvre un fichier en mode lecture seule en utilisant l'encodage détecté par `dectect_encoding()`.

Ajouté dans la version 3.2.

exception `tokenize.`**`TokenError`**

Déclenché lorsque soit une *docstring* soit une expression qui pourrait être divisée sur plusieurs lignes n'est pas complété dans le fichier, par exemple :

```
"""Beginning of
docstring
```

ou :

```
[1,
 2,
 3
```

33.5.2 Utilisation en ligne de commande.

Ajouté dans la version 3.3.

Le module `tokenize` peut être exécuté en tant que script à partir de la ligne de commande. C'est aussi simple que :

```
python -m tokenize [-e] [filename.py]
```

Les options suivantes sont acceptées :

`-h, --help`

Montre ce message d'aide et quitte

`-e, --exact`

Affiche les noms de jetons en utilisant le même type.

Si `filename.py` est spécifié, son contenu est tokenisé vers *stdout*. Sinon, la tokenisation est effectuée sur ce qui est fourni sur *stdin*.

33.5.3 Exemples

Exemple d'un script qui transforme les littéraux de type *float* en type *Decimal* :

```
from tokenize import tokenize, untokenize, NUMBER, STRING, NAME, OP
from io import BytesIO

def decistmt(s):
    """Substitute Decimals for floats in a string of statements.

    >>> from decimal import Decimal
    >>> s = 'print(+21.3e-5*-.1234/81.7)'
    >>> decistmt(s)
    "print (+Decimal ('21.3e-5')*-Decimal ('.1234')/Decimal ('81.7'))"
```

(suite sur la page suivante)

(suite de la page précédente)

```
    The format of the exponent is inherited from the platform C library.
    Known cases are "e-007" (Windows) and "e-07" (not Windows).  Since
    we're only showing 12 digits, and the 13th isn't close to 5, the
    rest of the output should be platform-independent.

    >>> exec(s)  #doctest: +ELLIPSIS
    -3.21716034272e-0...7

    Output from calculations with Decimal should be identical across all
    platforms.

    >>> exec(decistmt(s))
    -3.217160342717258261933904529E-7
    """
    result = []
    g = tokenize(BytesIO(s.encode('utf-8')).readline)  # tokenize the string
    for toknum, tokval, _, _, _ in g:
        if toknum == NUMBER and '.' in tokval:  # replace NUMBER tokens
            result.extend([
                (NAME, 'Decimal'),
                (OP, '('),
                (STRING, repr(tokval)),
                (OP, ')')
            ])
        else:
            result.append((toknum, tokval))
    return untokenize(result).decode('utf-8')
```

Exemple de tokenisation à partir de la ligne de commande. Le script :

```
def say_hello():
    print("Hello, World!")

say_hello()
```

sera tokenisé à la sortie suivante où la première colonne est la plage des coordonnées de la ligne/colonne où se trouve le jeton, la deuxième colonne est le nom du jeton, et la dernière colonne est la valeur du jeton (le cas échéant)

```
$ python -m tokenize hello.py
0,0-0,0:            ENCODING       'utf-8'
1,0-1,3:            NAME           'def'
1,4-1,13:           NAME           'say_hello'
1,13-1,14:          OP             '('
1,14-1,15:          OP             ')'
1,15-1,16:          OP             ':'
1,16-1,17:          NEWLINE        '\n'
2,0-2,4:            INDENT         '    '
2,4-2,9:            NAME           'print'
2,9-2,10:           OP             '('
2,10-2,25:          STRING         '"Hello, World!"'
2,25-2,26:          OP             ')'
2,26-2,27:          NEWLINE        '\n'
3,0-3,1:            NL             '\n'
4,0-4,0:            DEDENT         ''
4,0-4,9:            NAME           'say_hello'
4,9-4,10:           OP             '('
4,10-4,11:          OP             ')'
```

(suite sur la page suivante)

(suite de la page précédente)

```
4,11-4,12:          NEWLINE         '\n'
5,0-5,0:            ENDMARKER       ''
```

Les noms exacts des types de jeton peuvent être affichés en utilisant l'option : `-e` :

```
$ python -m tokenize -e hello.py
0,0-0,0:            ENCODING        'utf-8'
1,0-1,3:            NAME            'def'
1,4-1,13:           NAME            'say_hello'
1,13-1,14:          LPAR            '('
1,14-1,15:          RPAR            ')'
1,15-1,16:          COLON           ':'
1,16-1,17:          NEWLINE         '\n'
2,0-2,4:            INDENT          '    '
2,4-2,9:            NAME            'print'
2,9-2,10:           LPAR            '('
2,10-2,25:          STRING          '"Hello, World!"'
2,25-2,26:          RPAR            ')'
2,26-2,27:          NEWLINE         '\n'
3,0-3,1:            NL              '\n'
4,0-4,0:            DEDENT          ''
4,0-4,9:            NAME            'say_hello'
4,9-4,10:           LPAR            '('
4,10-4,11:          RPAR            ')'
4,11-4,12:          NEWLINE         '\n'
5,0-5,0:            ENDMARKER       ''
```

Example of tokenizing a file programmatically, reading unicode strings instead of bytes with `generate_tokens()` :

```
import tokenize

with tokenize.open('hello.py') as f:
    tokens = tokenize.generate_tokens(f.readline)
    for token in tokens:
        print(token)
```

Or reading bytes directly with `tokenize()` :

```
import tokenize

with open('hello.py', 'rb') as f:
    tokens = tokenize.tokenize(f.readline)
    for token in tokens:
        print(token)
```

33.6 `tabnanny` --- Detection of ambiguous indentation

Code source : Lib/tabnanny.py

Pour l'instant ce module est destiné à être appelé comme un script. Toutefois, il est possible de l'importer dans un IDE et d'utiliser la fonction `check()` décrite ci-dessous.

Note

L'API fournie par ce module est susceptible de changer dans les versions futures ; ces modifications peuvent ne pas être rétro-compatibles.

`tabnanny.check`(*file_or_dir*)

Si *file_or_dir* est un répertoire et non un lien symbolique, alors descend récursivement l'arborescence de répertoire nommé par *file_or_dir*, en vérifiant tous les fichiers `.py` en chemin. Si *file_or_dir* est un fichier source Python ordinaire, il est vérifié pour les problèmes liés aux espaces blancs. Les messages de diagnostic sont écrits sur la sortie standard à l'aide de la fonction `print()`.

`tabnanny.verbose`

Option indiquant s'il faut afficher des messages détaillés. Cela est incrémenté par l'option `-v` s'il est appelé comme un script.

`tabnanny.filename_only`

Option indiquant s'il faut afficher uniquement les noms de fichiers contenant des problèmes liés aux espaces blancs. Est défini à True par l'option `-q` s'il est appelé comme un script.

exception `tabnanny.NannyNag`

Déclenché par `process_tokens()` si une indentation ambiguë est détectée. Capturé et géré dans `check()`.

`tabnanny.process_tokens`(*tokens*)

Cette fonction est utilisée par `check()` pour traiter les jetons générés par le module `tokenize`.

Voir aussi

Module `tokenize`
Analyseur lexical pour le code source Python.

33.7 `pyclbr` --- Python module browser support

Code source : Lib/pyclbr.py

The `pyclbr` module provides limited information about the functions, classes, and methods defined in a Python-coded module. The information is sufficient to implement a module browser. The information is extracted from the Python source code rather than by importing the module, so this module is safe to use with untrusted code. This restriction makes it impossible to use this module with modules not implemented in Python, including all standard and optional extension modules.

`pyclbr.readmodule`(*module*, *path=None*)

Return a dictionary mapping module-level class names to class descriptors. If possible, descriptors for imported base classes are included. Parameter *module* is a string with the name of the module to read ; it may be the name of a module within a package. If given, *path* is a sequence of directory paths prepended to `sys.path`, which is used to locate the module source code.

This function is the original interface and is only kept for back compatibility. It returns a filtered version of the following.

`pyclbr.readmodule_ex`(*module*, *path=None*)

Return a dictionary-based tree containing a function or class descriptors for each function and class defined in the module with a `def` or `class` statement. The returned dictionary maps module-level function and class names to their descriptors. Nested objects are entered into the children dictionary of their parent. As with readmodule, *module* names the module to be read and *path* is prepended to sys.path. If the module being read is a package, the returned dictionary has a key `'__path__'` whose value is a list containing the package search path.

Ajouté dans la version 3.7 : Descriptors for nested definitions. They are accessed through the new children attribute. Each has a new parent attribute.

The descriptors returned by these functions are instances of Function and Class classes. Users are not expected to create instances of these classes.

33.7.1 Objets fonctions

class pyclbr.**Function**

Class `Function` instances describe functions defined by def statements. They have the following attributes :

file

Name of the file in which the function is defined.

module

The name of the module defining the function described.

name

The name of the function.

lineno

The line number in the file where the definition starts.

parent

For top-level functions, `None`. For nested functions, the parent.

Ajouté dans la version 3.7.

children

A *dictionary* mapping names to descriptors for nested functions and classes.

Ajouté dans la version 3.7.

is_async

`True` for functions that are defined with the `async` prefix, `False` otherwise.

Ajouté dans la version 3.10.

33.7.2 Objets classes

class pyclbr.**Class**

Class `Class` instances describe classes defined by class statements. They have the same attributes as *Functions* and two more.

file

Name of the file in which the class is defined.

module

The name of the module defining the class described.

name

The name of the class.

lineno

The line number in the file where the definition starts.

parent

For top-level classes, `None`. For nested classes, the parent.

Ajouté dans la version 3.7.

children

A dictionary mapping names to descriptors for nested functions and classes.

Ajouté dans la version 3.7.

super

A list of `Class` objects which describe the immediate base classes of the class being described. Classes which are named as superclasses but which are not discoverable by `readmodule_ex()` are listed as a string with the class name instead of as `Class` objects.

methods

A `dictionary` mapping method names to line numbers. This can be derived from the newer `children` dictionary, but remains for back-compatibility.

33.8 `py_compile` --- Compile Python source files

Code source : Lib/py_compile.py

Le module `py_compile` définit une fonction principale qui génère un fichier de code intermédiaire à partir d'un fichier source. Il exporte également la fonction qu'il exécute quand il est lancé en tant que script.

Bien que ce module ne soit pas d'usage fréquent, il peut servir lors de l'installation de bibliothèques partagées, notamment dans le cas où tous les utilisateurs n'ont pas les privilèges d'écriture dans l'emplacement d'installation.

exception py_compile.**PyCompileError**

Exception levée quand une erreur se produit à la compilation.

py_compile.**compile**(*file*, *cfile=None*, *dfile=None*, *doraise=False*, *optimize=-1*, *invalidation_mode=PycInvalidationMode.TIMESTAMP*, *quiet=0*)

Compile a source file to byte-code and write out the byte-code cache file. The source code is loaded from the file named *file*. The byte-code is written to *cfile*, which defaults to the **PEP 3147/PEP 488** path, ending in `.pyc`. For example, if *file* is `/foo/bar/baz.py` *cfile* will default to `/foo/bar/__pycache__/baz.cpython-32.pyc` for Python 3.2. If *dfile* is specified, it is used instead of *file* as the name of the source file from which source lines are obtained for display in exception tracebacks. If *doraise* is true, a `PyCompileError` is raised when an error is encountered while compiling *file*. If *doraise* is false (the default), an error string is written to `sys.stderr`, but no exception is raised. This function returns the path to byte-compiled file, i.e. whatever *cfile* value was used.

Plus précisément, ce sont les arguments *doraise* et *quiet* qui déterminent la stratégie de gestion des erreurs. En effet, si *quiet* vaut 0 ou 1, et *doraise* est faux, le comportement par défaut s'applique : un message d'erreur est affiché dans `sys.stderr`, et la fonction renvoie `None` au lieu d'un chemin. Au contraire, si *doraise* est vrai, une exception `PyCompileError` est levée, sauf si *quiet* vaut 2. Dans ce dernier cas, aucun message n'est émis, et *doraise* reste sans effet.

Si le chemin de destination, explicité par *cfile* ou choisi automatiquement, est un lien symbolique, ou n'est pas un véritable fichier, une exception de type `FileExistsError` est levée. Ceci, dans le but de vous avertir que le système d'importation changera ces chemins en fichiers s'il est autorisé à y écrire des fichiers de code intermédiaire. En effet, les importations passent par un renommage final du fichier de code intermédiaire vers sa destination, afin d'éviter les problèmes liés à l'écriture simultanée d'un même fichier par plusieurs processus.

optimize règle le niveau d'optimisation. Ce paramètre est passé directement à la fonction native `compile()`. Avec la valeur par défaut de `-1`, le code intermédiaire hérite du niveau d'optimisation de l'interpréteur courant.

invalidation_mode précise la manière dont le code intermédiaire produit est invalidé à son exécution. Il doit être un membre de l'énumération `PycInvalidationMode`. La valeur par défaut est `PycInvalidationMode.TIMESTAMP`. Elle passe toutefois à `PycInvalidationMode.CHECKED_HASH` si la variable d'environnement `SOURCE_DATE_EPOCH` est définie.

Modifié dans la version 3.2 : la méthode de choix de destination a changé au profit de celle décrite dans la **PEP 3147**. Auparavant, le nom du fichier de code intermédiaire était *file* + `'c'` (ou `'o'` lorsque les optimisations étaient actives). Le paramètre *optimize* a été ajouté.

Modifié dans la version 3.4 : le code a été modifié pour faire appel à `importlib` dans les opérations d'écriture du code intermédiaire. Ce module se comporte donc exactement comme `importlib` en ce qui concerne, par exemple, les permissions, ou le renommage final qui garantit une opération atomique. `FileExistsError` est désormais levée si la destination est un lien symbolique ou n'est pas un véritable fichier.

Modifié dans la version 3.7 : le paramètre *invalidation_mode* a été ajouté comme requis par la **PEP 552**. Si la variable d'environnement `SOURCE_DATE_EPOCH` est définie, *invalidation_mode* est ignoré, et `PycInvalidationMode.CHECKED_HASH` s'applique dans tous les cas.

Modifié dans la version 3.7.2 : La variable d'environnement `SOURCE_DATE_EPOCH` n'a plus préséance sur le paramètre *invalidation_mode*, mais détermine seulement le comportement par défaut lorsque ce paramètre n'est pas passé.

Modifié dans la version 3.8 : ajout du paramètre *quiet*.

class `py_compile.`**`PycInvalidationMode`**

An enumeration of possible methods the interpreter can use to determine whether a bytecode file is up to date with a source file. The `.pyc` file indicates the desired invalidation mode in its header. See pyc-invalidation for more information on how Python invalidates `.pyc` files at runtime.

Ajouté dans la version 3.7.

`TIMESTAMP`

Le fichier `.pyc` contient l'horodatage et la taille de la source. L'interpréteur inspecte les métadonnées du fichier source au moment de l'exécution, et régénère le fichier `.pyc` si elles ont changé.

`CHECKED_HASH`

Le fichier `.pyc` porte une empreinte du code source. À l'exécution, elle est recalculée à partir de la source éventuellement modifiée, et le fichier `.pyc` est régénéré si les deux empreintes sont différentes.

`UNCHECKED_HASH`

Le principe est le même que `CHECKED_HASH`, mais à l'exécution, l'interpréteur considère systématiquement que le fichier `.pyc` est à jour, sans regarder la source.

Cette option est utile lorsque les fichiers `.pyc` sont maintenus par un outil externe, comme un système d'intégration.

33.8.1 Interface en ligne de commande

This module can be invoked as a script to compile several source files. The files named in *filenames* are compiled and the resulting bytecode is cached in the normal manner. This program does not search a directory structure to locate source files ; it only compiles files named explicitly. The exit status is nonzero if one of the files could not be compiled.

`<file>` `... <fileN>`

`-`

Positional arguments are files to compile. If `-` is the only parameter, the list of files is taken from standard input.

`-q, --quiet`

Suppress errors output.

Modifié dans la version 3.2 : Added support for `-`.

Modifié dans la version 3.10 : Added support for `-q`.

Voir aussi

Module `compileall`
Utilitaires pour compiler des fichiers source Python dans une arborescence

33.9 `compileall` --- Byte-compile Python libraries

Code source : Lib/compileall.py

Ce module contient des fonctions qui facilitent l'installation de bibliothèques Python. Elles compilent, sous forme de code intermédiaire (*bytecode*), les fichiers source situés dans un dossier de votre choix. Ce module est particulièrement utile pour générer les fichiers de code intermédiaire lors de l'installation d'une bibliothèque, les rendant disponibles même pour les utilisateurs qui n'ont pas les privilèges d'écriture dans l'emplacement d'installation.

Availability : not WASI.

This module does not work or is not available on WebAssembly. See *Plateformes WebAssembly* for more information.

33.9.1 Utilisation en ligne de commande

On peut se servir de ce module comme d'un script (avec **`python -m compileall`**) pour compiler les fichiers source Python.

`directory ...`
`file ...`

Positional arguments are files to compile or directories that contain source files, traversed recursively. If no argument is given, behave as if the command line was `-l <directories from sys.path>`.

`-l`

Compiler uniquement les fichiers situés directement dans les dossiers passés en argument ou implicites, sans descendre récursivement dans les sous-dossiers.

`-f`

Forcer la recompilation même si les horodatages sont à jour.

`-q`

Supprimer l'affichage des noms des fichiers compilés.Si cette option est donnée une seule fois, les erreurs sont affichées malgré tout. Vous pouvez les supprimer en passant l'option deux fois (c'est-à-dire avec `-qq`).

`-d` `destdir`

Ce nom de dossier est ajouté en tête du chemin de chaque fichier compilé. Il aura une influence sur les traces d'appels pour les erreurs levées lors de la compilation, et sera reflété dans les fichiers de code intermédiaire, pour utilisation dans les traces d'appels et autres messages si le fichier source n'existe pas au moment de l'exécution.

`-s` `strip_prefix`

`-p` `prepend_prefix`

Retire (`-s`) ou ajoute (`-p`) le préfixe aux chemins stockés dans les fichiers `.pyc`. Cette option ne peut pas être combinée avec `-d`.

`-x` `regex`

Exclut tous les fichiers dont les noms correspondent à l'expression régulière *regex*.

`-i` `list`

Ajoute chaque ligne du fichier *list* aux fichiers et dossiers à compiler. *list* peut être `-`, auquel cas le script lit l'entrée standard.

`-b`

Utilise l'ancienne manière de nommer et placer les fichiers de code intermédiaire, en écrasant éventuellement ceux générés par une autre version de Python. Par défaut, les règles décrites dans la **PEP 3147** s'appliquent. Elles permettent à différentes versions de l'interpréteur Python de coexister en conservant chacune ses propres fichiers `.pyc`.

-r

Règle le niveau de récursion maximal pour le parcours des sous-dossiers. Lorsque cette option est fournie, `-l` est ignorée. **python -m compileall <dossier> -r 0** revient au même que **python -m compileall <dossier> -l**.

-j N

Use *N* workers to compile the files within the given directory. If `0` is used, then the result of `os.process_cpu_count()` will be used.

--invalidation-mode [timestamp|checked-hash|unchecked-hash]

Définit la manière dont les fichiers de code intermédiaire seront invalidés au moment de l'exécution. Avec `timestamp`, les fichiers `.pyc` générés comportent l'horodatage de la source et sa taille. Avec `checked-hash` ou `unchecked-hash`, ce seront des pyc utilisant le hachage, qui contiennent une empreinte du code source plutôt qu'un horodatage. Voir pyc-invalidation pour plus d'informations sur la manière dont Python valide les fichiers de code intermédiaire conservés en cache lors de l'exécution. La valeur par défaut est `timestamp`. Cependant, si la variable d'environnement `SOURCE_DATE_EPOCH` a été réglée, elle devient `checked-hash`.

-o level

Compile avec un certain niveau d'optimisation. Cette option peut être passée plusieurs fois afin de compiler pour plusieurs niveaux d'un seul coup (par exemple, `compileall -o 1 -o 2`).

-e dir

Ignore les liens symboliques qui redirigent en dehors du dossier.

--hardlink-dupes

Si deux fichiers `.pyc` compilés avec des niveaux d'optimisation différents ont finalement le même contenu, emploie des liens physiques pour les fusionner.

Modifié dans la version 3.2 : ajout des options `-i`, `-b` et `-h`.

Modifié dans la version 3.5 : ajout des options `-j`, `-r` et `-qq` (l'option `-q` peut donc prendre plusieurs niveaux). `-b` produit toujours un fichier de code intermédiaire portant l'extension `.pyc`, et jamais `.pyo`.

Modifié dans la version 3.7 : ajout de l'option `--invalidation-mode`.

Modifié dans la version 3.9 : ajout des options `-s`, `-p`, `-e` et `--hardlink-dupes`. Rehaussement de la limite de récursion par défaut à `sys.getrecursionlimit()` au lieu de 10 précédemment. L'option `-o` peut être passée plusieurs fois.

Il n'y a pas d'option en ligne de commande pour contrôler le niveau d'optimisation utilisé par la fonction `compile()`. Il suffit en effet d'utiliser l'option `-O` de l'interpréteur Python lui-même : **python -O -m compileall**.

Similarly, the `compile()` function respects the `sys.pycache_prefix` setting. The generated bytecode cache will only be useful if `compile()` is run with the same `sys.pycache_prefix` (if any) that will be used at runtime.

33.9.2 Fonctions publiques

compileall.**compile_dir**(*dir*, *maxlevels=sys.getrecursionlimit()*, *ddir=None*, *force=False*, *rx=None*, *quiet=0*, *legacy=False*, *optimize=-1*, *workers=1*, *invalidation_mode=None*, ***, *stripdir=None*, *prependdir=None*, *limit_sl_dest=None*, *hardlink_dupes=False*)

Parcourt récursivement le dossier *dir*, en compilant tous les fichiers `.py`. Renvoie une valeur vraie si tous les fichiers ont été compilés sans erreur, et une valeur fausse dans le cas contraire.

Le paramètre *maxlevels* permet de limiter la profondeur de récursion. Sa valeur par défaut est celle de `sys.getrecursionlimit()`.

Si *ddir* est fourni, il est ajouté en tête du chemin de chaque fichier compilé, ce qui modifie l'affichage des traces d'appels pour les erreurs qui seraient levées lors de la compilation. De plus, il se retrouve dans les fichiers de code intermédiaire, pour utilisation dans les traces et autres messages si le fichier source n'existe pas au moment de l'exécution.

Si *force* est vrai, les modules sont recompilés même si leurs horodatages sont à jour.

Si *rx* est donné, sa méthode `search` est appelée sur le chemin complet de chaque fichier source, et si elle renvoie une valeur vraie, le fichier est sauté. *rx* sera habituellement une expression régulière (objet *re.Pattern*).

Si *quiet* est `False` ou bien `0` (la valeur par défaut), les noms de fichiers et d'autres informations sont affichés sur la sortie standard. Avec `1`, seules les erreurs sont affichées. Avec `2`, aucune sortie n'est émise.

Si *legacy* est vrai, les fichiers de code intermédiaire sont nommés et placés selon l'ancienne méthode, en écrasant éventuellement ceux générés par une autre version de Python. Par défaut, les règles décrites dans la **PEP 3147** s'appliquent. Elles permettent à différentes versions de l'interpréteur Python de coexister en conservant chacune ses propres fichiers `.pyc`.

optimize définit le niveau d'optimisation qu'applique le compilateur. Cet argument est passé directement à la fonction native `compile()`. Il peut également être fourni sous la forme d'une séquence de niveaux d'optimisation, ce qui permet de compiler chaque fichier `.py` plusieurs fois en appliquant divers niveaux d'optimisation.

workers est le nombre de tâches lancées en parallèle pour la compilation. Par défaut, les fichiers sont compilés séquentiellement. Cette même stratégie s'applique dans tous les cas lorsque le parallélisme n'est pas possible sur la plateforme d'exécution. Si *workers* vaut 0, autant de tâches sont lancées que le système comporte de cœurs. Si *workers* est strictement négatif, une exception de type `ValueError` est levée.

invalidation_mode doit être un membre de l'énumération `py_compile.PycInvalidationMode` et détermine la manière dont les fichiers `.pyc` sont invalidés lorsque l'interpréteur tente de les utiliser.

The *stripdir*, *prependdir* and *limit_sl_dest* arguments correspond to the `-s`, `-p` and `-e` options described above. They may be specified as `str` or `os.PathLike`.

Un argument *hardlink_dupes* vrai correspond à l'utilisation de l'option `--hardlink-dupes`.

Modifié dans la version 3.2 : ajout des paramètres *legacy* et *optimize*.

Modifié dans la version 3.5 : ajout du paramètre *workers*.

Modifié dans la version 3.5 : le paramètre *quiet* peut prendre plusieurs niveaux.

Modifié dans la version 3.5 : Lorsque le paramètre *legacy* est vrai, des fichiers `.pyc`, et jamais `.pyo`, sont générés, quel que soit le niveau d'optimisation.

Modifié dans la version 3.6 : accepte un *objet simili-chemin*.

Modifié dans la version 3.7 : ajout du paramètre *invalidation_mode*.

Modifié dans la version 3.7.2 : The *invalidation_mode* parameter's default value is updated to `None`.

Modifié dans la version 3.8 : Un nombre de processus adapté à la machine est choisi lorsque *workers* vaut 0.

Modifié dans la version 3.9 : ajout des arguments *stripdir*, *prependdir*, *limit_sl_dest* et *hardlink_dupes*. La valeur par défaut de *maxlevels* a été changée pour `sys.getrecursionlimit()` (elle était de 10 auparavant).

`compileall.`**`compile_file`**(*fullname*, *ddir=None*, *force=False*, *rx=None*, *quiet=0*, *legacy=False*, *optimize=-1*, *invalidation_mode=None*, ***, *stripdir=None*, *prependdir=None*, *limit_sl_dest=None*, *hardlink_dupes=False*)

Compile le fichier dont le chemin est donné par *fullname*. Renvoie une valeur vraie si et seulement si le fichier est compilé sans erreur.

Si *ddir* est fourni, il est ajouté en tête du chemin de chaque fichier compilé, ce qui modifie l'affichage des traces pour les erreurs qui seraient levées lors de la compilation. De plus, il se retrouve dans les fichiers de code intermédiaire, pour utilisation dans les traces et autres messages si le fichier source n'existe pas au moment de l'exécution.

Si *rx* est donné, sa méthode `search` est appelée sur le chemin complet de chaque fichier source, et si elle renvoie une valeur vraie, le fichier est sauté. *rx* sera habituellement une expression régulière (objet *re.Pattern*).

Si *quiet* est `False` ou bien `0` (la valeur par défaut), les noms de fichiers et d'autres informations sont affichés sur la sortie standard. Avec `1`, seules les erreurs sont affichées. Avec `2`, aucune sortie n'est émise.

Si *legacy* est vrai, les fichiers de code intermédiaire sont nommés et placés selon l'ancienne méthode, en écrasant éventuellement ceux générés par une autre version de Python. Par défaut, les règles décrites dans la **PEP 3147** s'appliquent. Elles permettent à différentes versions de l'interpréteur Python de coexister en conservant chacune

ses propres fichiers `.pyc`.

optimize définit le niveau d'optimisation qu'applique le compilateur. Cet argument est passé directement à la fonction native `compile()`. Il peut également être fourni sous la forme d'une séquence de niveaux d'optimisation, ce qui permet de compiler chaque fichier `.py` plusieurs fois en appliquant divers niveaux d'optimisation.

invalidation_mode doit être un membre de l'énumération `py_compile.PycInvalidationMode` et détermine la manière dont les fichiers `.pyc` sont invalidés lorsque l'interpréteur tente de les utiliser.

The *stripdir*, *prependdir* and *limit_sl_dest* arguments correspond to the `-s`, `-p` and `-e` options described above. They may be specified as `str` or `os.PathLike`.

Un argument *hardlink_dupes* vrai correspond à l'utilisation de l'option `--hardlink-dupes`.

Ajouté dans la version 3.2.

Modifié dans la version 3.5 : le paramètre *quiet* peut prendre plusieurs niveaux.

Modifié dans la version 3.5 : Lorsque le paramètre *legacy* est vrai, des fichiers `.pyc`, et jamais `.pyo`, sont générés, quel que soit le niveau d'optimisation.

Modifié dans la version 3.7 : ajout du paramètre *invalidation_mode*.

Modifié dans la version 3.7.2 : The *invalidation_mode* parameter's default value is updated to `None`.

Modifié dans la version 3.9 : Ajout des arguments *stripdir*, *prependdir*, *limit_sl_dest* et *hardlink_dupes*.

`compileall.`**`compile_path`**(*skip_curdir=True*, *maxlevels=0*, *force=False*, *quiet=0*, *legacy=False*, *optimize=-1*, *invalidation_mode=None*)

Compile tous les fichiers `.py` contenus dans les dossiers de `sys.path`. Renvoie une valeur vraie s'ils ont tous été compilés sans erreur, et une valeur fausse dans le cas contraire.

Si *skip_curdir* est vrai (c'est le cas par défaut), le dossier courant est exclu de la recherche. Les autres paramètres sont passés à `compile_dir()`. Notez que contrairement aux autres fonctions de ce module, la valeur par défaut de `maxlevels` est `0`.

Modifié dans la version 3.2 : ajout des paramètres *legacy* et *optimize*.

Modifié dans la version 3.5 : le paramètre *quiet* peut prendre plusieurs niveaux.

Modifié dans la version 3.5 : Lorsque le paramètre *legacy* est vrai, des fichiers `.pyc`, et jamais `.pyo`, sont générés, quel que soit le niveau d'optimisation.

Modifié dans la version 3.7 : ajout du paramètre *invalidation_mode*.

Modifié dans la version 3.7.2 : The *invalidation_mode* parameter's default value is updated to `None`.

Pour forcer la recompilation de tous les fichiers `.py` dans le dossier `Lib/` et tous ses sous-dossiers :

```
import compileall

compileall.compile_dir('Lib/', force=True)

# Perform same compilation, excluding files in .svn directories.
import re
compileall.compile_dir('Lib/', rx=re.compile(r'[/\\][.]svn'), force=True)

# pathlib.Path objects can also be used.
import pathlib
compileall.compile_dir(pathlib.Path('Lib/'), force=True)
```

Voir aussi

Module `py_compile`
Compiler un fichier source unique.

33.10 dis --- Disassembler for Python bytecode

Code source : Lib/dis.py

La bibliothèque *dis* supporte l'analyse du *bytecode* CPython en le désassemblant. Le code intermédiaire CPython, que cette bibliothèque prend en paramètre, est défini dans le fichier `Include/opcode.h` et est utilisé par le compilateur et l'interpréteur.

Le code intermédiaire est un détail d'implémentation de l'interpréteur CPython. Il n'y a pas de garantie que le code intermédiaire sera ajouté, retiré, ou modifié dans les différentes versions de Python. L'utilisation de cette bibliothèque ne fonctionne pas nécessairement sur les machines virtuelles Python ni les différentes versions de Python.

Modifié dans la version 3.6 : Utilisez 2 bits pour chaque instruction. Avant, le nombre de bits variait par instruction.

Modifié dans la version 3.10 : The argument of jump, exception handling and loop instructions is now the instruction offset rather than the byte offset.

Modifié dans la version 3.11 : Some instructions are accompanied by one or more inline cache entries, which take the form of *CACHE* instructions. These instructions are hidden by default, but can be shown by passing `show_caches=True` to any *dis* utility. Furthermore, the interpreter now adapts the bytecode to specialize it for different runtime conditions. The adaptive bytecode can be shown by passing `adaptive=True`.

Modifié dans la version 3.12 : The argument of a jump is the offset of the target instruction relative to the instruction that appears immediately after the jump instruction's *CACHE* entries.

As a consequence, the presence of the *CACHE* instructions is transparent for forward jumps but needs to be taken into account when reasoning about backward jumps.

Modifié dans la version 3.13 : The output shows logical labels rather than instruction offsets for jump targets and exception handlers. The `-O` command line option and the `show_offsets` argument were added.

Modifié dans la version 3.14 : The *-P* command-line option and the `show_positions` argument were added.

The *-S* command-line option is added.

Example : Given the function `myfunc()` :

```
def myfunc(alist):
    return len(alist)
```

the following command can be used to display the disassembly of `myfunc()` :

```
>>> dis.dis(myfunc)
  2           RESUME                   0

  3           LOAD_GLOBAL              1 (len + NULL)
              LOAD_FAST                0 (alist)
              CALL                     1
              RETURN_VALUE
```

(Le "2" est un numéro de ligne).

33.10.1 Command-line interface

The *dis* module can be invoked as a script from the command line :

```
python -m dis [-h] [-C] [-O] [-P] [-S] [infile]
```

Les options suivantes sont acceptées :

-h, --help

Display usage and exit.

-C, --show-caches

Show inline caches.

-O, --show-offsets

Show offsets of instructions.

-P, --show-positions

Show positions of instructions in the source code.

-S, --specialized

Show specialized bytecode.

If `infile` is specified, its disassembled code will be written to stdout. Otherwise, disassembly is performed on compiled source code received from stdin.

33.10.2 Analyse du code intermédiaire

Ajouté dans la version 3.4.

L'analyse de l'*API* code intermédiaire permet de rassembler des blocs de code en Python dans une classe `Bytecode`, qui permet un accès facile aux détails du code compilé.

class dis.**Bytecode**(*x*, ***, *first_line=None*, *current_offset=None*, *show_caches=False*, *adaptive=False*, *show_offsets=False*, *show_positions=False*)

Analyse le code intermédiaire correspondant à une fonction, un générateur, un générateur asynchrone, une coroutine, une méthode, une chaîne de caractères du code source, ou bien une classe (comme retourne la fonction `compile()`).

Ceci est *wrapper* sur plusieurs fonctions de la liste ci-dessous, notamment `get_instructions()`, étant donné qu'une itération sur une instance de la classe `Bytecode` rend les opérations du code intermédiaire des instances de `Instruction`.

Si *first_line* ne vaut pas `None`, elle indique le nombre de la ligne qui doit être considérée comme première ligne source dans le code désassemblé. Autrement, les informations sur la ligne source sont prises directement à partir de la classe du code désassemblé.

Si la valeur de *current_offset* est différente de `None`, c'est une référence à un offset d'une instruction dans le code désassemblé. Cela veut dire que `dis()` va générer un marqueur de " l'instruction en cours" contre le code d'opération donné.

If *show_caches* is `True`, `dis()` will display inline cache entries used by the interpreter to specialize the bytecode.

If *adaptive* is `True`, `dis()` will display specialized bytecode that may be different from the original bytecode.

If *show_offsets* is `True`, `dis()` will include instruction offsets in the output.

If *show_positions* is `True`, `dis()` will include instruction source code positions in the output.

classmethod from_traceback(*tb*, ***, *show_caches=False*)

Construisez une instance `Bytecode` à partir de la trace d'appel, en mettant *current_offet* à l'instruction responsable de l'exception.

codeobj

Le code compilé objet.

first_line

La première ligne source du code objet (si disponible)

dis()

Retourne une vue formatée des opérations du code intermédiaire (la même que celle envoyée par `dis.dis()`, mais comme une chaîne de caractères de plusieurs lignes).

info()

Retourne une chaîne de caractères de plusieurs lignes formatée avec des informations détaillées sur l'objet code comme `code_info()`.

Modifié dans la version 3.7 : Cette version supporte la coroutine et les objets générateurs asynchrones.

Modifié dans la version 3.11 : Added the *show_caches* and *adaptive* parameters.

Modifié dans la version 3.13 : Added the *show_offsets* parameter

Modifié dans la version 3.14 : Added the *show_positions* parameter.

Example :

```
>>> bytecode = dis.Bytecode(myfunc)
>>> for instr in bytecode:
...     print(instr.opname)
...
RESUME
LOAD_GLOBAL
LOAD_FAST
CALL
RETURN_VALUE
```

33.10.3 Analyse de fonctions

La bibliothèque `dis` comprend également l'analyse des fonctions suivantes, qui envoient l'entrée directement à la sortie souhaitée. Elles peuvent être utiles si il n'y a qu'une seule opération à effectuer, la représentation intermédiaire objet n'étant donc pas utile dans ce cas :

dis.**code_info**(*x*)

Retourne une chaîne de caractères de plusieurs lignes formatée avec des informations détaillées sur l'objet code pour les fonctions données, les générateurs asynchrone, coroutine, la méthode, la chaine de caractères du code source ou objet.

Il est à noter que le contenu exact des chaînes de caractères figurant dans les informations du code dépendent fortement sur l'implémentation, et peuvent changer arbitrairement sous machines virtuelles Python ou les versions de Python.

Ajouté dans la version 3.2.

Modifié dans la version 3.7 : Cette version supporte la coroutine et les objets générateurs asynchrones.

dis.**show_code**(*x*, *, *file=None*)

Affiche des informations détaillées sur le code de la fonction fournie, la méthode, la chaîne de caractère du code source ou du code objet à *file* (ou bien `sys.stdout` si *file* n'est pas spécifié).

Ceci est un raccourci convenable de `print(code_info(x), file=file)`, principalement fait pour l'exploration interactive sur l'invite de l'interpréteur.

Ajouté dans la version 3.2.

Modifié dans la version 3.4 : Ajout du paramètre *file*.

dis.**dis**(*x=None*, *, *file=None*, *depth=None*, *show_caches=False*, *adaptive=False*, *show_offsets=False*, *show_positions=False*)

Disassemble the *x* object. *x* can denote either a module, a class, a method, a function, a generator, an asynchronous generator, a coroutine, a code object, a string of source code or a byte sequence of raw bytecode. For a module, it disassembles all functions. For a class, it disassembles all methods (including class and static methods). For a code object or sequence of raw bytecode, it prints one line per bytecode instruction. It also recursively disassembles nested code objects. These can include generator expressions, nested functions, the bodies of nested classes, and the code objects used for annotation scopes. Strings are first compiled to code objects with the `compile()` built-in function before being disassembled. If no object is provided, this function disassembles the last traceback.

Le désassemblage est envoyé sous forme de texte à l'argument du fichier *file* si il est fourni, et à `sys.stdout` sinon.

La profondeur maximale de récursion est limitée par *depth* sauf si elle correspond à `None`. `depth=0` indique qu'il n'y a pas de récursion.

If *show_caches* is `True`, this function will display inline cache entries used by the interpreter to specialize the bytecode.

If *adaptive* is `True`, this function will display specialized bytecode that may be different from the original bytecode.

Modifié dans la version 3.4 : Ajout du paramètre *file*.

Modifié dans la version 3.7 : Le désassemblage récursif a été implémenté, et le paramètre *depth* a été ajouté.

Modifié dans la version 3.7 : Cette version supporte la coroutine et les objets générateurs asynchrones.

Modifié dans la version 3.11 : Added the *show_caches* and *adaptive* parameters.

Modifié dans la version 3.13 : Added the *show_offsets* parameter.

Modifié dans la version 3.14 : Added the *show_positions* parameter.

`dis.`**`distb`**(*tb=None*, ***, *file=None*, *show_caches=False*, *adaptive=False*, *show_offset=False*, *show_positions=False*)

Désassemble la fonction du haut de la pile des traces d'appels, en utilisant la dernière trace d'appels si rien n'a été envoyé. L'instruction à l'origine de l'exception est indiquée.

Le désassemblage est envoyé sous forme de texte à l'argument du fichier *file* si il est fourni, et à `sys.stdout` sinon.

Modifié dans la version 3.4 : Ajout du paramètre *file*.

Modifié dans la version 3.11 : Added the *show_caches* and *adaptive* parameters.

Modifié dans la version 3.13 : Added the *show_offsets* parameter.

Modifié dans la version 3.14 : Added the *show_positions* parameter.

`dis.`**`disassemble`**(*code*, *lasti=-1*, ***, *file=None*, *show_caches=False*, *adaptive=False*, *show_offsets=False*, *show_positions=False*)

`dis.`**`disco`**(*code*, *lasti=-1*, ***, *file=None*, *show_caches=False*, *adaptive=False*, *show_offsets=False*, *show_positions=False*)

Désassemble un code objet, en indiquant la dernière instruction si *lasti* est fournie. La sortie est répartie sur les colonnes suivantes :

1. the source code location of the instruction. Complete location information is shown if *show_positions* is true. Otherwise (the default) only the line number is displayed.
2. l'instruction en cours, indiquée par `-->`,
3. une instruction libellée, indiquée par `>>`,
4. l'adresse de l'instruction,
5. le nom de le code d'opération,
6. paramètres de l'opération, et
7. interprétation des paramètres entre parenthèses.

L'interprétation du paramètre reconnaît les noms des variables locales et globales, des valeurs constantes, des branchements cibles, et des opérateurs de comparaison.

Le désassemblage est envoyé sous forme de texte à l'argument du fichier *file* si il est fourni, et à `sys.stdout` sinon.

Modifié dans la version 3.4 : Ajout du paramètre *file*.

Modifié dans la version 3.11 : Added the *show_caches* and *adaptive* parameters.

Modifié dans la version 3.13 : Added the *show_offsets* parameter.

Modifié dans la version 3.14 : Added the *show_positions* parameter.

dis.**get_instructions**(*x*, *, *first_line=None*, *show_caches=False*, *adaptive=False*)

Retourne un itérateur sur les instructions dans la fonction fournie, la méthode, les chaînes de caractères du code source ou objet.

Cet itérateur génère une série de *n*-uplets de `Instruction` qui donnent les détails de chacune des opérations dans le code fourni.

Si *first_line* ne vaut pas `None`, elle indique le nombre de la ligne qui doit être considérée comme première ligne source dans le code désassemblé. Autrement, les informations sur la ligne source sont prises directement à partir de la classe du code désassemblé.

The *adaptive* parameter works as it does in `dis()`.

Ajouté dans la version 3.4.

Modifié dans la version 3.11 : Added the *show_caches* and *adaptive* parameters.

Modifié dans la version 3.13 : The *show_caches* parameter is deprecated and has no effect. The iterator generates the `Instruction` instances with the *cache_info* field populated (regardless of the value of *show_caches*) and it no longer generates separate items for the cache entries.

dis.**findlinestarts**(*code*)

This generator function uses the `co_lines()` method of the code object *code* to find the offsets which are starts of lines in the source code. They are generated as `(offset, lineno)` pairs.

Modifié dans la version 3.6 : Les numéros de lignes peuvent être décroissants. Avant, ils étaient toujours croissants.

Modifié dans la version 3.10 : The **PEP 626** `co_lines()` method is used instead of the `co_firstlineno` and `co_lnotab` attributes of the code object.

Modifié dans la version 3.13 : Line numbers can be `None` for bytecode that does not map to source lines.

dis.**findlabels**(*code*)

Detect all offsets in the raw compiled bytecode string *code* which are jump targets, and return a list of these offsets.

dis.**stack_effect**(*opcode*, *oparg=None*, *, *jump=None*)

Compute the stack effect of *opcode* with argument *oparg*.

If the code has a jump target and *jump* is `True`, `stack_effect()` will return the stack effect of jumping. If *jump* is `False`, it will return the stack effect of not jumping. And if *jump* is `None` (default), it will return the maximal stack effect of both cases.

Ajouté dans la version 3.4.

Modifié dans la version 3.8 : Added *jump* parameter.

Modifié dans la version 3.13 : If `oparg` is omitted (or `None`), the stack effect is now returned for `oparg=0`. Previously this was an error for opcodes that use their arg. It is also no longer an error to pass an integer `oparg` when the `opcode` does not use it ; the `oparg` in this case is ignored.

33.10.4 Les instructions du code intermédiaire en Python

La fonction `get_instructions()` et la méthode `Bytecode` fournit des détails sur le code intermédiaire des instructions comme `Instruction` instances :

class dis.**Instruction**

Détails sur le code intermédiaire de l'opération

opcode

code numérique pour l'opération, correspondant aux valeurs de l'*opcode* ci-dessous et les valeurs du code intermédiaire dans la *Opcode collections*.

opname

nom lisible/compréhensible de l'opération

baseopcode

numeric code for the base operation if operation is specialized ; otherwise equal to *opcode*

baseopname

human readable name for the base operation if operation is specialized ; otherwise equal to *opname*

arg

le cas échéant, argument numérique de l'opération sinon `None`

oparg

alias for *arg*

argval

resolved arg value (if any), otherwise `None`

argrepr

human readable description of operation argument (if any), otherwise an empty string.

offset

start index of operation within bytecode sequence

start_offset

start index of operation within bytecode sequence, including prefixed `EXTENDED_ARG` operations if present ; otherwise equal to *offset*

cache_offset

start index of the cache entries following the operation

end_offset

end index of the cache entries following the operation

starts_line

`True` if this opcode starts a source line, otherwise `False`

line_number

source line number associated with this opcode (if any), otherwise `None`

is_jump_target

`True` if other code jumps to here, otherwise `False`

jump_target

bytecode index of the jump target if this is a jump operation, otherwise `None`

positions

dis.Positions object holding the start and end locations that are covered by this instruction.

Ajouté dans la version 3.4.

Modifié dans la version 3.11 : Field `positions` is added.

Modifié dans la version 3.13 : Changed field `starts_line`.

Added fields `start_offset`, `cache_offset`, `end_offset`, `baseopname`, `baseopcode`, `jump_target`, `oparg`, `line_number` and `cache_info`.

class dis.**Positions**

In case the information is not available, some fields might be `None`.

lineno

end_lineno

col_offset

end_col_offset

Ajouté dans la version 3.11.

The Python compiler currently generates the following bytecode instructions.

General instructions

In the following, We will refer to the interpreter stack as `STACK` and describe operations on it as if it was a Python list. The top of the stack corresponds to `STACK[-1]` in this language.

NOP

Do nothing code. Used as a placeholder by the bytecode optimizer, and to generate line tracing events.

POP_TOP

Removes the top-of-stack item :

```
STACK.pop()
```

END_FOR

Removes the top-of-stack item. Equivalent to `POP_TOP`. Used to clean up at the end of loops, hence the name.

Ajouté dans la version 3.12.

END_SEND

Implements `del STACK[-2]`. Used to clean up when a generator exits.

Ajouté dans la version 3.12.

COPY (*i*)

Push the i-th item to the top of the stack without removing it from its original location :

```
assert i > 0
STACK.append(STACK[-i])
```

Ajouté dans la version 3.11.

SWAP (*i*)

Swap the top of the stack with the i-th element :

```
STACK[-i], STACK[-1] = STACK[-1], STACK[-i]
```

Ajouté dans la version 3.11.

CACHE

Rather than being an actual instruction, this opcode is used to mark extra space for the interpreter to cache useful data directly in the bytecode itself. It is automatically hidden by all `dis` utilities, but can be viewed with `show_caches=True`.

Logically, this space is part of the preceding instruction. Many opcodes expect to be followed by an exact number of caches, and will instruct the interpreter to skip over them at runtime.

Populated caches can look like arbitrary instructions, so great care should be taken when reading or modifying raw, adaptive bytecode containing quickened data.

Ajouté dans la version 3.11.

Unary operations

Unary operations take the top of the stack, apply the operation, and push the result back on the stack.

UNARY_NEGATIVE

Implements `STACK[-1] = -STACK[-1]`.

UNARY_NOT

Implements `STACK[-1] = not STACK[-1]`.

Modifié dans la version 3.13 : This instruction now requires an exact `bool` operand.

UNARY_INVERT

Implements `STACK[-1] = ~STACK[-1]`.

GET_ITER

Implements `STACK[-1] = iter(STACK[-1])`.

GET_YIELD_FROM_ITER

If `STACK[-1]` is a *generator iterator* or *coroutine* object it is left as is. Otherwise, implements `STACK[-1] = iter(STACK[-1])`.

Ajouté dans la version 3.5.

TO_BOOL

Implements `STACK[-1] = bool(STACK[-1])`.

Ajouté dans la version 3.13.

Binary and in-place operations

Binary operations remove the top two items from the stack (`STACK[-1]` and `STACK[-2]`). They perform the operation, then put the result back on the stack.

In-place operations are like binary operations, but the operation is done in-place when `STACK[-2]` supports it, and the resulting `STACK[-1]` may be (but does not have to be) the original `STACK[-2]`.

BINARY_OP (*op*)

Implements the binary and in-place operators (depending on the value of *op*) :

```
rhs = STACK.pop()
lhs = STACK.pop()
STACK.append(lhs op rhs)
```

Ajouté dans la version 3.11.

Modifié dans la version 3.14 : With oparg `:NB_SUBSCR`, implements binary subscript (replaces opcode `BINARY_SUBSCR`)

STORE_SUBSCR

Implements :

```
key = STACK.pop()
container = STACK.pop()
value = STACK.pop()
container[key] = value
```

DELETE_SUBSCR

Implements :

```
key = STACK.pop()
container = STACK.pop()
del container[key]
```

BINARY_SLICE

Implements :

```
end = STACK.pop()
start = STACK.pop()
```

(suite sur la page suivante)

(suite de la page précédente)

```
container = STACK.pop()
STACK.append(container[start:end])
```

Ajouté dans la version 3.12.

STORE_SLICE

Implements :

```
end = STACK.pop()
start = STACK.pop()
container = STACK.pop()
values = STACK.pop()
container[start:end] = value
```

Ajouté dans la version 3.12.

Coroutine opcodes

GET_AWAITABLE (*where*)

Implements `STACK[-1] = get_awaitable(STACK[-1])`, where `get_awaitable(o)` returns `o` if `o` is a coroutine object or a generator object with the *CO_ITERABLE_COROUTINE* flag, or resolves `o.__await__`.

If the `where` operand is nonzero, it indicates where the instruction occurs :

— `1` : After a call to `__aenter__`
— `2` : After a call to `__aexit__`

Ajouté dans la version 3.5.

Modifié dans la version 3.11 : Previously, this instruction did not have an oparg.

GET_AITER

Implements `STACK[-1] = STACK[-1].__aiter__()`.

Ajouté dans la version 3.5.

Modifié dans la version 3.7 : Returning awaitable objects from `__aiter__` is no longer supported.

GET_ANEXT

Implement `STACK.append(get_awaitable(STACK[-1].__anext__()))` to the stack. See `GET_AWAITABLE` for details about `get_awaitable`.

Ajouté dans la version 3.5.

END_ASYNC_FOR

Terminates an `async for` loop. Handles an exception raised when awaiting a next item. The stack contains the async iterable in `STACK[-2]` and the raised exception in `STACK[-1]`. Both are popped. If the exception is not *StopAsyncIteration*, it is re-raised.

Ajouté dans la version 3.8.

Modifié dans la version 3.11 : Exception representation on the stack now consist of one, not three, items.

CLEANUP_THROW

Handles an exception raised during a `throw()` or `close()` call through the current frame. If `STACK[-1]` is an instance of *StopIteration*, pop three values from the stack and push its `value` member. Otherwise, re-raise `STACK[-1]`.

Ajouté dans la version 3.12.

Miscellaneous opcodes

SET_ADD (*i*)

Implements :

```
item = STACK.pop()
set.add(STACK[-i], item)
```

Used to implement set comprehensions.

LIST_APPEND (*i*)

Implements :

```
item = STACK.pop()
list.append(STACK[-i], item)
```

Used to implement list comprehensions.

MAP_ADD (*i*)

Implements :

```
value = STACK.pop()
key = STACK.pop()
dict.__setitem__(STACK[-i], key, value)
```

Used to implement dict comprehensions.

Ajouté dans la version 3.1.

Modifié dans la version 3.8 : Map value is `STACK[-1]` and map key is `STACK[-2]`. Before, those were reversed.

For all of the *SET_ADD*, *LIST_APPEND* and *MAP_ADD* instructions, while the added value or key/value pair is popped off, the container object remains on the stack so that it is available for further iterations of the loop.

RETURN_VALUE

Returns with `STACK[-1]` to the caller of the function.

YIELD_VALUE

Yields `STACK.pop()` from a *generator*.

Modifié dans la version 3.11 : oparg set to be the stack depth.

Modifié dans la version 3.12 : oparg set to be the exception block depth, for efficient closing of generators.

Modifié dans la version 3.13 : oparg is `1` if this instruction is part of a yield-from or await, and `0` otherwise.

SETUP_ANNOTATIONS

Checks whether `__annotations__` is defined in `locals()`, if not it is set up to an empty `dict`. This opcode is only emitted if a class or module body contains *variable annotations* statically.

Ajouté dans la version 3.6.

POP_EXCEPT

Pops a value from the stack, which is used to restore the exception state.

Modifié dans la version 3.11 : Exception representation on the stack now consist of one, not three, items.

RERAISE

Re-raises the exception currently on top of the stack. If oparg is non-zero, pops an additional value from the stack which is used to set `f_lasti` of the current frame.

Ajouté dans la version 3.9.

Modifié dans la version 3.11 : Exception representation on the stack now consist of one, not three, items.

PUSH_EXC_INFO

Pops a value from the stack. Pushes the current exception to the top of the stack. Pushes the value originally popped back to the stack. Used in exception handlers.

Ajouté dans la version 3.11.

CHECK_EXC_MATCH

Performs exception matching for `except`. Tests whether the `STACK[-2]` is an exception matching `STACK[-1]`. Pops `STACK[-1]` and pushes the boolean result of the test.

Ajouté dans la version 3.11.

CHECK_EG_MATCH

Performs exception matching for `except*`. Applies `split(STACK[-1])` on the exception group representing `STACK[-2]`.

In case of a match, pops two items from the stack and pushes the non-matching subgroup (`None` in case of full match) followed by the matching subgroup. When there is no match, pops one item (the match type) and pushes `None`.

Ajouté dans la version 3.11.

WITH_EXCEPT_START

Calls the function in position 4 on the stack with arguments (type, val, tb) representing the exception at the top of the stack. Used to implement the call `context_manager.__exit__(*exc_info())` when an exception has occurred in a `with` statement.

Ajouté dans la version 3.9.

Modifié dans la version 3.11 : The `__exit__` function is in position 4 of the stack rather than 7. Exception representation on the stack now consist of one, not three, items.

LOAD_COMMON_CONSTANT

Pushes a common constant onto the stack. The interpreter contains a hardcoded list of constants supported by this instruction. Used by the `assert` statement to load *`AssertionError`*.

Ajouté dans la version 3.14.

LOAD_BUILD_CLASS

Pushes `builtins.__build_class__()` onto the stack. It is later called to construct a class.

GET_LEN

Perform `STACK.append(len(STACK[-1]))`. Used in `match` statements where comparison with structure of pattern is needed.

Ajouté dans la version 3.10.

MATCH_MAPPING

If `STACK[-1]` is an instance of *`collections.abc.Mapping`* (or, more technically : if it has the `Py_TPFLAGS_MAPPING` flag set in its `tp_flags`), push `True` onto the stack. Otherwise, push `False`.

Ajouté dans la version 3.10.

MATCH_SEQUENCE

If `STACK[-1]` is an instance of *`collections.abc.Sequence`* and is *not* an instance of *`str`*/*`bytes`*/*`bytearray`* (or, more technically : if it has the `Py_TPFLAGS_SEQUENCE` flag set in its `tp_flags`), push `True` onto the stack. Otherwise, push `False`.

Ajouté dans la version 3.10.

MATCH_KEYS

`STACK[-1]` is a tuple of mapping keys, and `STACK[-2]` is the match subject. If `STACK[-2]` contains all of the keys in `STACK[-1]`, push a *`tuple`* containing the corresponding values. Otherwise, push `None`.

Ajouté dans la version 3.10.

Modifié dans la version 3.11 : Previously, this instruction also pushed a boolean value indicating success (`True`) or failure (`False`).

STORE_NAME (*namei*)

Implements `name = STACK.pop()`. *namei* is the index of *name* in the attribute `co_names` of the code object. The compiler tries to use *`STORE_FAST`* or *`STORE_GLOBAL`* if possible.

DELETE_NAME (*namei*)

Implements `del name`, where *namei* is the index into `co_names` attribute of the code object.

UNPACK_SEQUENCE (*count*)

Unpacks `STACK[-1]` into *count* individual values, which are put onto the stack right-to-left. Require there to be exactly *count* values. :

```
assert(len(STACK[-1]) == count)
STACK.extend(STACK.pop()[:-count-1:-1])
```

UNPACK_EX (*counts*)

Implements assignment with a starred target : Unpacks an iterable in `STACK[-1]` into individual values, where the total number of values can be smaller than the number of items in the iterable : one of the new values will be a list of all leftover items.

The number of values before and after the list value is limited to 255.

The number of values before the list value is encoded in the argument of the opcode. The number of values after the list if any is encoded using an `EXTENDED_ARG`. As a consequence, the argument can be seen as a two bytes values where the low byte of *counts* is the number of values before the list value, the high byte of *counts* the number of values after it.

The extracted values are put onto the stack right-to-left, i.e. `a, *b, c = d` will be stored after execution as `STACK.extend((a, b, c))`.

STORE_ATTR (*namei*)

Implements :

```
obj = STACK.pop()
value = STACK.pop()
obj.name = value
```

where *namei* is the index of name in `co_names` of the code object.

DELETE_ATTR (*namei*)

Implements :

```
obj = STACK.pop()
del obj.name
```

where *namei* is the index of name into `co_names` of the code object.

STORE_GLOBAL (*namei*)

Works as *STORE_NAME*, but stores the name as a global.

DELETE_GLOBAL (*namei*)

Works as *DELETE_NAME*, but deletes a global name.

LOAD_CONST (*consti*)

Pushes `co_consts[consti]` onto the stack.

LOAD_SMALL_INT (*i*)

Pushes the integer `i` onto the stack. `i` must be in `range(256)`

Ajouté dans la version 3.14.

LOAD_CONST_IMMORTAL (*consti*)

Pushes `co_consts[consti]` onto the stack. Can be used when the constant value is known to be immortal.

Ajouté dans la version 3.14.

LOAD_NAME (*namei*)

Pushes the value associated with `co_names[namei]` onto the stack. The name is looked up within the locals, then the globals, then the builtins.

LOAD_LOCALS

Pushes a reference to the locals dictionary onto the stack. This is used to prepare namespace dictionaries for *LOAD_FROM_DICT_OR_DEREF* and *LOAD_FROM_DICT_OR_GLOBALS*.

Ajouté dans la version 3.12.

LOAD_FROM_DICT_OR_GLOBALS (*i*)

Pops a mapping off the stack and looks up the value for `co_names[namei]`. If the name is not found there, looks it up in the globals and then the builtins, similar to *LOAD_GLOBAL*. This is used for loading global variables in annotation scopes within class bodies.

Ajouté dans la version 3.12.

BUILD_TUPLE (*count*)

Creates a tuple consuming *count* items from the stack, and pushes the resulting tuple onto the stack :

```
if count == 0:
    value = ()
else:
    value = tuple(STACK[-count:])
    STACK = STACK[:-count]

STACK.append(value)
```

BUILD_LIST (*count*)

Works as *BUILD_TUPLE*, but creates a list.

BUILD_SET (*count*)

Works as *BUILD_TUPLE*, but creates a set.

BUILD_MAP (*count*)

Pushes a new dictionary object onto the stack. Pops `2 * count` items so that the dictionary holds *count* entries : `{..., STACK[-4]: STACK[-3], STACK[-2]: STACK[-1]}`.

Modifié dans la version 3.5 : The dictionary is created from stack items instead of creating an empty dictionary pre-sized to hold *count* items.

BUILD_STRING (*count*)

Concatenates *count* strings from the stack and pushes the resulting string onto the stack.

Ajouté dans la version 3.6.

LIST_EXTEND (*i*)

Implements :

```
seq = STACK.pop()
list.extend(STACK[-i], seq)
```

Used to build lists.

Ajouté dans la version 3.9.

SET_UPDATE (*i*)

Implements :

```
seq = STACK.pop()
set.update(STACK[-i], seq)
```

Used to build sets.

Ajouté dans la version 3.9.

DICT_UPDATE (*i*)

Implements :

```
map = STACK.pop()
dict.update(STACK[-i], map)
```

Used to build dicts.

Ajouté dans la version 3.9.

DICT_MERGE (*i*)

Like *DICT_UPDATE* but raises an exception for duplicate keys.

Ajouté dans la version 3.9.

LOAD_ATTR (*namei*)

If the low bit of `namei` is not set, this replaces `STACK[-1]` with `getattr(STACK[-1], co_names[namei>>1])`.

If the low bit of `namei` is set, this will attempt to load a method named `co_names[namei>>1]` from the `STACK[-1]` object. `STACK[-1]` is popped. This bytecode distinguishes two cases : if `STACK[-1]` has a method with the correct name, the bytecode pushes the unbound method and `STACK[-1]`. `STACK[-1]` will be used as the first argument (`self`) by *CALL* or *CALL_KW* when calling the unbound method. Otherwise, `NULL` and the object returned by the attribute lookup are pushed.

Modifié dans la version 3.12 : If the low bit of `namei` is set, then a `NULL` or `self` is pushed to the stack before the attribute or unbound method respectively.

LOAD_SUPER_ATTR (*namei*)

This opcode implements *super()*, both in its zero-argument and two-argument forms (e.g. `super().method()`, `super().attr` and `super(cls, self).method()`, `super(cls, self).attr`).

It pops three values from the stack (from top of stack down) :
— `self` : the first argument to the current method
— `cls` : the class within which the current method was defined
— the global `super`
With respect to its argument, it works similarly to *LOAD_ATTR*, except that `namei` is shifted left by 2 bits instead of 1.

The low bit of `namei` signals to attempt a method load, as with *LOAD_ATTR*, which results in pushing `NULL` and the loaded method. When it is unset a single value is pushed to the stack.

The second-low bit of `namei`, if set, means that this was a two-argument call to *super()* (unset means zero-argument).

Ajouté dans la version 3.12.

COMPARE_OP (*opname*)

Performs a Boolean operation. The operation name can be found in `cmp_op[opname >> 5]`. If the fifth-lowest bit of `opname` is set (`opname & 16`), the result should be coerced to `bool`.

Modifié dans la version 3.13 : The fifth-lowest bit of the oparg now indicates a forced conversion to *bool*.

IS_OP (*invert*)

Performs `is` comparison, or `is not` if `invert` is 1.

Ajouté dans la version 3.9.

CONTAINS_OP (*invert*)

Performs `in` comparison, or `not in` if `invert` is 1.

Ajouté dans la version 3.9.

IMPORT_NAME (*namei*)

Imports the module `co_names[namei]`. `STACK[-1]` and `STACK[-2]` are popped and provide the *fromlist*

and *level* arguments of `__import__()`. The module object is pushed onto the stack. The current namespace is not affected : for a proper import statement, a subsequent `STORE_FAST` instruction modifies the namespace.

IMPORT_FROM (*namei*)

Loads the attribute `co_names[namei]` from the module found in `STACK[-1]`. The resulting object is pushed onto the stack, to be subsequently stored by a `STORE_FAST` instruction.

JUMP_FORWARD (*delta*)

Increments bytecode counter by *delta*.

JUMP_BACKWARD (*delta*)

Decrements bytecode counter by *delta*. Checks for interrupts.

Ajouté dans la version 3.11.

JUMP_BACKWARD_NO_INTERRUPT (*delta*)

Decrements bytecode counter by *delta*. Does not check for interrupts.

Ajouté dans la version 3.11.

POP_JUMP_IF_TRUE (*delta*)

If `STACK[-1]` is true, increments the bytecode counter by *delta*. `STACK[-1]` is popped.

Modifié dans la version 3.11 : The oparg is now a relative delta rather than an absolute target. This opcode is a pseudo-instruction, replaced in final bytecode by the directed versions (forward/backward).

Modifié dans la version 3.12 : This is no longer a pseudo-instruction.

Modifié dans la version 3.13 : This instruction now requires an exact `bool` operand.

POP_JUMP_IF_FALSE (*delta*)

If `STACK[-1]` is false, increments the bytecode counter by *delta*. `STACK[-1]` is popped.

Modifié dans la version 3.11 : The oparg is now a relative delta rather than an absolute target. This opcode is a pseudo-instruction, replaced in final bytecode by the directed versions (forward/backward).

Modifié dans la version 3.12 : This is no longer a pseudo-instruction.

Modifié dans la version 3.13 : This instruction now requires an exact `bool` operand.

POP_JUMP_IF_NOT_NONE (*delta*)

If `STACK[-1]` is not `None`, increments the bytecode counter by *delta*. `STACK[-1]` is popped.

This opcode is a pseudo-instruction, replaced in final bytecode by the directed versions (forward/backward).

Ajouté dans la version 3.11.

Modifié dans la version 3.12 : This is no longer a pseudo-instruction.

POP_JUMP_IF_NONE (*delta*)

If `STACK[-1]` is `None`, increments the bytecode counter by *delta*. `STACK[-1]` is popped.

This opcode is a pseudo-instruction, replaced in final bytecode by the directed versions (forward/backward).

Ajouté dans la version 3.11.

Modifié dans la version 3.12 : This is no longer a pseudo-instruction.

FOR_ITER (*delta*)

`STACK[-1]` is an *iterator*. Call its `__next__()` method. If this yields a new value, push it on the stack (leaving the iterator below it). If the iterator indicates it is exhausted then the byte code counter is incremented by *delta*.

Modifié dans la version 3.12 : Up until 3.11 the iterator was popped when it was exhausted.

LOAD_GLOBAL (*namei*)

Loads the global named `co_names[namei>>1]` onto the stack.

Modifié dans la version 3.11 : If the low bit of `namei` is set, then a `NULL` is pushed to the stack before the global variable.

LOAD_FAST (*var_num*)

Pushes a reference to the local `co_varnames[var_num]` onto the stack.

Modifié dans la version 3.12 : This opcode is now only used in situations where the local variable is guaranteed to be initialized. It cannot raise `UnboundLocalError`.

LOAD_FAST_LOAD_FAST (*var_nums*)

Pushes references to `co_varnames[var_nums >> 4]` and `co_varnames[var_nums & 15]` onto the stack.

Ajouté dans la version 3.13.

LOAD_FAST_CHECK (*var_num*)

Pushes a reference to the local `co_varnames[var_num]` onto the stack, raising an `UnboundLocalError` if the local variable has not been initialized.

Ajouté dans la version 3.12.

LOAD_FAST_AND_CLEAR (*var_num*)

Pushes a reference to the local `co_varnames[var_num]` onto the stack (or pushes `NULL` onto the stack if the local variable has not been initialized) and sets `co_varnames[var_num]` to `NULL`.

Ajouté dans la version 3.12.

STORE_FAST (*var_num*)

Stores `STACK.pop()` into the local `co_varnames[var_num]`.

STORE_FAST_STORE_FAST (*var_nums*)

Stores `STACK[-1]` into `co_varnames[var_nums >> 4]` and `STACK[-2]` into `co_varnames[var_nums & 15]`.

Ajouté dans la version 3.13.

STORE_FAST_LOAD_FAST (*var_nums*)

Stores `STACK.pop()` into the local `co_varnames[var_nums >> 4]` and pushes a reference to the local `co_varnames[var_nums & 15]` onto the stack.

Ajouté dans la version 3.13.

DELETE_FAST (*var_num*)

Deletes local `co_varnames[var_num]`.

MAKE_CELL (*i*)

Creates a new cell in slot `i`. If that slot is nonempty then that value is stored into the new cell.

Ajouté dans la version 3.11.

LOAD_DEREF (*i*)

Loads the cell contained in slot `i` of the "fast locals" storage. Pushes a reference to the object the cell contains on the stack.

Modifié dans la version 3.11 : `i` is no longer offset by the length of `co_varnames`.

LOAD_FROM_DICT_OR_DEREF (*i*)

Pops a mapping off the stack and looks up the name associated with slot `i` of the "fast locals" storage in this mapping. If the name is not found there, loads it from the cell contained in slot `i`, similar to `LOAD_DEREF`. This is used for loading *closure variables* in class bodies (which previously used `LOAD_CLASSDEREF`) and in annotation scopes within class bodies.

Ajouté dans la version 3.12.

STORE_DEREF (*i*)

Stores `STACK.pop()` into the cell contained in slot `i` of the "fast locals" storage.

Modifié dans la version 3.11 : `i` is no longer offset by the length of `co_varnames`.

DELETE_DEREF (*i*)

Empties the cell contained in slot `i` of the "fast locals" storage. Used by the `del` statement.

Ajouté dans la version 3.2.

Modifié dans la version 3.11 : `i` is no longer offset by the length of `co_varnames`.

COPY_FREE_VARS (*n*)

Copies the `n` *free (closure) variables* from the closure into the frame. Removes the need for special code on the caller's side when calling closures.

Ajouté dans la version 3.11.

RAISE_VARARGS (*argc*)

Raises an exception using one of the 3 forms of the `raise` statement, depending on the value of *argc* :
— 0 : `raise` (re-raise previous exception)
— 1 : `raise STACK[-1]` (raise exception instance or type at `STACK[-1]`)
— 2 : `raise STACK[-2] from STACK[-1]` (raise exception instance or type at `STACK[-2]` with `__cause__` set to `STACK[-1]`)

CALL (*argc*)

Calls a callable object with the number of arguments specified by `argc`. On the stack are (in ascending order) :
— The callable
— `self` or `NULL`
— The remaining positional arguments
`argc` is the total of the positional arguments, excluding `self`.

`CALL` pops all arguments and the callable object off the stack, calls the callable object with those arguments, and pushes the return value returned by the callable object.

Ajouté dans la version 3.11.

Modifié dans la version 3.13 : The callable now always appears at the same position on the stack.

Modifié dans la version 3.13 : Calls with keyword arguments are now handled by *CALL_KW*.

CALL_KW (*argc*)

Calls a callable object with the number of arguments specified by `argc`, including one or more named arguments. On the stack are (in ascending order) :
— The callable
— `self` or `NULL`
— The remaining positional arguments
— The named arguments
— A *tuple* of keyword argument names
`argc` is the total of the positional and named arguments, excluding `self`. The length of the tuple of keyword argument names is the number of named arguments.

`CALL_KW` pops all arguments, the keyword names, and the callable object off the stack, calls the callable object with those arguments, and pushes the return value returned by the callable object.

Ajouté dans la version 3.13.

CALL_FUNCTION_EX (*flags*)

Calls a callable object with variable set of positional and keyword arguments. If the lowest bit of *flags* is set, the top of the stack contains a mapping object containing additional keyword arguments. Before the callable is called, the mapping object and iterable object are each "unpacked" and their contents passed in as keyword and positional arguments respectively. `CALL_FUNCTION_EX` pops all arguments and the callable object off the stack, calls the callable object with those arguments, and pushes the return value returned by the callable object.

Ajouté dans la version 3.6.

PUSH_NULL

Pushes a `NULL` to the stack. Used in the call sequence to match the `NULL` pushed by *LOAD_METHOD* for non-method calls.

Ajouté dans la version 3.11.

MAKE_FUNCTION

Pushes a new function object on the stack built from the code object at `STACK[-1]`.

Modifié dans la version 3.10 : Flag value `0x04` is a tuple of strings instead of dictionary

Modifié dans la version 3.11 : Qualified name at `STACK[-1]` was removed.

Modifié dans la version 3.13 : Extra function attributes on the stack, signaled by oparg flags, were removed. They now use *SET_FUNCTION_ATTRIBUTE*.

SET_FUNCTION_ATTRIBUTE(*flag*)

Sets an attribute on a function object. Expects the function at `STACK[-1]` and the attribute value to set at `STACK[-2]`; consumes both and leaves the function at `STACK[-1]`. The flag determines which attribute to set :

- `0x01` a tuple of default values for positional-only and positional-or-keyword parameters in positional order
- `0x02` a dictionary of keyword-only parameters' default values
- `0x04` a tuple of strings containing parameters' annotations
- `0x08` a tuple containing cells for free variables, making a closure

Ajouté dans la version 3.13.

BUILD_SLICE(*argc*)

Pushes a slice object on the stack. *argc* must be 2 or 3. If it is 2, implements :

```
end = STACK.pop()
start = STACK.pop()
STACK.append(slice(start, end))
```

if it is 3, implements :

```
step = STACK.pop()
end = STACK.pop()
start = STACK.pop()
STACK.append(slice(start, end, step))
```

See the *slice()* built-in function for more information.

EXTENDED_ARG(*ext*)

Prefixes any opcode which has an argument too big to fit into the default one byte. *ext* holds an additional byte which act as higher bits in the argument. For each opcode, at most three prefixal `EXTENDED_ARG` are allowed, forming an argument from two-byte to four-byte.

CONVERT_VALUE(*oparg*)

Convert value to a string, depending on `oparg` :

```
value = STACK.pop()
result = func(value)
STACK.append(result)
```

- `oparg == 1` : call *str()* on *value*
- `oparg == 2` : call *repr()* on *value*
- `oparg == 3` : call *ascii()* on *value*

Used for implementing formatted literal strings (f-strings).

Ajouté dans la version 3.13.

FORMAT_SIMPLE

Formats the value on top of stack :

```
value = STACK.pop()
result = value.__format__("")
STACK.append(result)
```

Used for implementing formatted literal strings (f-strings).

Ajouté dans la version 3.13.

FORMAT_WITH_SPEC

Formats the given value with the given format spec :

```
spec = STACK.pop()
value = STACK.pop()
result = value.__format__(spec)
STACK.append(result)
```

Used for implementing formatted literal strings (f-strings).

Ajouté dans la version 3.13.

MATCH_CLASS (*count*)

`STACK[-1]` is a tuple of keyword attribute names, `STACK[-2]` is the class being matched against, and `STACK[-3]` is the match subject. *count* is the number of positional sub-patterns.

Pop `STACK[-1]`, `STACK[-2]`, and `STACK[-3]`. If `STACK[-3]` is an instance of `STACK[-2]` and has the positional and keyword attributes required by *count* and `STACK[-1]`, push a tuple of extracted attributes. Otherwise, push `None`.

Ajouté dans la version 3.10.

Modifié dans la version 3.11 : Previously, this instruction also pushed a boolean value indicating success (`True`) or failure (`False`).

RESUME (*context*)

A no-op. Performs internal tracing, debugging and optimization checks.

The `context` operand consists of two parts. The lowest two bits indicate where the `RESUME` occurs :
— `0` The start of a function, which is neither a generator, coroutine nor an async generator
— `1` After a `yield` expression
— `2` After a `yield from` expression
— `3` After an `await` expression
The next bit is `1` if the RESUME is at except-depth `1`, and `0` otherwise.

Ajouté dans la version 3.11.

Modifié dans la version 3.13 : The oparg value changed to include information about except-depth

RETURN_GENERATOR

Create a generator, coroutine, or async generator from the current frame. Used as first opcode of in code object for the above mentioned callables. Clear the current frame and return the newly created generator.

Ajouté dans la version 3.11.

SEND (*delta*)

Equivalent to `STACK[-1] = STACK[-2].send(STACK[-1])`. Used in `yield from` and `await` statements.

If the call raises `StopIteration`, pop the top value from the stack, push the exception's `value` attribute, and increment the bytecode counter by *delta*.

Ajouté dans la version 3.11.

HAVE_ARGUMENT

This is not really an opcode. It identifies the dividing line between opcodes in the range [0,255] which don't use their argument and those that do (`< HAVE_ARGUMENT` and `>= HAVE_ARGUMENT`, respectively).

If your application uses pseudo instructions or specialized instructions, use the *hasarg* collection instead.

Modifié dans la version 3.6 : Now every instruction has an argument, but opcodes `< HAVE_ARGUMENT` ignore it. Before, only opcodes `>= HAVE_ARGUMENT` had an argument.

Modifié dans la version 3.12 : Pseudo instructions were added to the *dis* module, and for them it is not true that comparison with `HAVE_ARGUMENT` indicates whether they use their arg.

Obsolète depuis la version 3.13 : Use *hasarg* instead.

CALL_INTRINSIC_1

Calls an intrinsic function with one argument. Passes `STACK[-1]` as the argument and sets `STACK[-1]` to the result. Used to implement functionality that is not performance critical.

The operand determines which intrinsic function is called :

Operand	Description
`INTRINSIC_1_INVAL`	Not valid
`INTRINSIC_PRINT`	Prints the argument to standard out. Used in the REPL.
`INTRINSIC_IMPORT_`	Performs `import *` for the named module.
`INTRINSIC_STOPITE`	Extracts the return value from a `StopIteration` exception.
`INTRINSIC_ASYNC_G`	Wraps an async generator value
`INTRINSIC_UNARY_P`	Performs the unary + operation
`INTRINSIC_LIST_TO_`	Converts a list to a tuple
`INTRINSIC_TYPEVAR`	Creates a *typing.TypeVar*
`INTRINSIC_PARAMSP`	Creates a *typing.ParamSpec*
`INTRINSIC_TYPEVAR`	Creates a *typing.TypeVarTuple*
`INTRINSIC_SUBSCRI`	Returns *typing.Generic* subscripted with the argument
`INTRINSIC_TYPEALI`	Creates a *typing.TypeAliasType*; used in the `type` statement. The argument is a tuple of the type alias's name, type parameters, and value.

Ajouté dans la version 3.12.

CALL_INTRINSIC_2

Calls an intrinsic function with two arguments. Used to implement functionality that is not performance critical :

```
arg2 = STACK.pop()
arg1 = STACK.pop()
result = intrinsic2(arg1, arg2)
STACK.append(result)
```

The operand determines which intrinsic function is called :

Operand	Description
`INTRINSIC_2_INVALID`	Not valid
`INTRINSIC_PREP_RERAISE_STAR`	Calculates the *ExceptionGroup* to raise from a `try-except*`.
`INTRINSIC_TYPEVAR_WITH_BOUND`	Creates a *typing.TypeVar* with a bound.
`INTRINSIC_TYPEVAR_WITH_CONSTRAIN`	Creates a *typing.TypeVar* with constraints.
`INTRINSIC_SET_FUNCTION_TYPE_PARA`	Sets the `__type_params__` attribute of a function.

Ajouté dans la version 3.12.

LOAD_SPECIAL

Performs special method lookup on `STACK[-1]`. If `type(STACK[-1]).__xxx__` is a method, leave `type(STACK[-1]).__xxx__; STACK[-1]` on the stack. If `type(STACK[-1]).__xxx__` is not a method, leave `STACK[-1].__xxx__; NULL` on the stack.

Ajouté dans la version 3.14.

Pseudo-instructions

These opcodes do not appear in Python bytecode. They are used by the compiler but are replaced by real opcodes or removed before bytecode is generated.

SETUP_FINALLY (*target*)

Set up an exception handler for the following code block. If an exception occurs, the value stack level is restored to its current state and control is transferred to the exception handler at `target`.

SETUP_CLEANUP (*target*)

Like `SETUP_FINALLY`, but in case of an exception also pushes the last instruction (`lasti`) to the stack so that `RERAISE` can restore it. If an exception occurs, the value stack level and the last instruction on the frame are restored to their current state, and control is transferred to the exception handler at `target`.

SETUP_WITH (*target*)

Like `SETUP_CLEANUP`, but in case of an exception one more item is popped from the stack before control is transferred to the exception handler at `target`.

This variant is used in `with` and `async with` constructs, which push the return value of the context manager's `__enter__()` or `__aenter__()` to the stack.

POP_BLOCK

Marks the end of the code block associated with the last `SETUP_FINALLY`, `SETUP_CLEANUP` or `SETUP_WITH`.

JUMP

JUMP_NO_INTERRUPT

Undirected relative jump instructions which are replaced by their directed (forward/backward) counterparts by the assembler.

JUMP_IF_TRUE

JUMP_IF_FALSE

Conditional jumps which do not impact the stack. Replaced by the sequence `COPY 1`, `TO_BOOL`, `POP_JUMP_IF_TRUE/FALSE`.

LOAD_CLOSURE (*i*)

Pushes a reference to the cell contained in slot `i` of the "fast locals" storage.

Note that `LOAD_CLOSURE` is replaced with `LOAD_FAST` in the assembler.

Modifié dans la version 3.13 : This opcode is now a pseudo-instruction.

LOAD_METHOD

Optimized unbound method lookup. Emitted as a `LOAD_ATTR` opcode with a flag set in the arg.

33.10.5 Opcode collections

These collections are provided for automatic introspection of bytecode instructions :

Modifié dans la version 3.12 : The collections now contain pseudo instructions and instrumented instructions as well. These are opcodes with values `>= MIN_PSEUDO_OPCODE` and `>= MIN_INSTRUMENTED_OPCODE`.

`dis.`**opname**

Sequence of operation names, indexable using the bytecode.

`dis.`**opmap**

Dictionary mapping operation names to bytecodes.

`dis.`**cmp_op**

Sequence of all compare operation names.

`dis.hasarg`

Sequence of bytecodes that use their argument.

Ajouté dans la version 3.12.

`dis.hasconst`

Sequence of bytecodes that access a constant.

`dis.hasfree`

Sequence of bytecodes that access a *free (closure) variable*. 'free' in this context refers to names in the current scope that are referenced by inner scopes or names in outer scopes that are referenced from this scope. It does *not* include references to global or builtin scopes.

`dis.hasname`

Sequence of bytecodes that access an attribute by name.

`dis.hasjump`

Sequence of bytecodes that have a jump target. All jumps are relative.

Ajouté dans la version 3.13.

`dis.haslocal`

Sequence of bytecodes that access a local variable.

`dis.hascompare`

Sequence of bytecodes of Boolean operations.

`dis.hasexc`

Sequence of bytecodes that set an exception handler.

Ajouté dans la version 3.12.

`dis.hasjrel`

Sequence of bytecodes that have a relative jump target.

Obsolète depuis la version 3.13 : All jumps are now relative. Use `hasjump`.

`dis.hasjabs`

Sequence of bytecodes that have an absolute jump target.

Obsolète depuis la version 3.13 : All jumps are now relative. This list is empty.

33.11 `pickletools` --- Tools for pickle developers

Source code : Lib/pickletools.py

This module contains various constants relating to the intimate details of the `pickle` module, some lengthy comments about the implementation, and a few useful functions for analyzing pickled data. The contents of this module are useful for Python core developers who are working on the `pickle`; ordinary users of the `pickle` module probably won't find the `pickletools` module relevant.

33.11.1 Utilisation de la ligne de commande

Ajouté dans la version 3.2.

When invoked from the command line, `python -m pickletools` will disassemble the contents of one or more pickle files. Note that if you want to see the Python object stored in the pickle rather than the details of pickle format, you may want to use `-m pickle` instead. However, when the pickle file that you want to examine comes from an untrusted source, `-m pickletools` is a safer option because it does not execute pickle bytecode.

For example, with a tuple `(1, 2)` pickled in file `x.pickle` :

```
$ python -m pickle x.pickle
(1, 2)

$ python -m pickletools x.pickle
    0: \x80 PROTO      3
    2: K    BININT1    1
    4: K    BININT1    2
    6: \x86 TUPLE2
    7: q    BINPUT     0
    9: .    STOP
highest protocol among opcodes = 2
```

Options de la ligne de commande

-a, --annotate

Annotate each line with a short opcode description.

-o, --output=<file>

Name of a file where the output should be written.

-l, --indentlevel=<num>

The number of blanks by which to indent a new MARK level.

-m, --memo

When multiple objects are disassembled, preserve memo between disassemblies.

-p, --preamble=<preamble>

When more than one pickle file are specified, print given preamble before each disassembly.

33.11.2 Programmatic Interface

`pickletools.`**`dis`**(*pickle*, *out=None*, *memo=None*, *indentlevel=4*, *annotate=0*)

Outputs a symbolic disassembly of the pickle to the file-like object *out*, defaulting to `sys.stdout`. *pickle* can be a string or a file-like object. *memo* can be a Python dictionary that will be used as the pickle's memo ; it can be used to perform disassemblies across multiple pickles created by the same pickler. Successive levels, indicated by `MARK` opcodes in the stream, are indented by *indentlevel* spaces. If a nonzero value is given to *annotate*, each opcode in the output is annotated with a short description. The value of *annotate* is used as a hint for the column where annotation should start.

Modifié dans la version 3.2 : Added the *annotate* parameter.

`pickletools.`**`genops`**(*pickle*)

Provides an *iterator* over all of the opcodes in a pickle, returning a sequence of `(opcode, arg, pos)` triples. *opcode* is an instance of an `OpcodeInfo` class ; *arg* is the decoded value, as a Python object, of the opcode's argument ; *pos* is the position at which this opcode is located. *pickle* can be a string or a file-like object.

`pickletools.`**`optimize`**(*picklestring*)

Returns a new equivalent pickle string after eliminating unused `PUT` opcodes. The optimized pickle is shorter, takes less transmission time, requires less storage space, and unpickles more efficiently.

CHAPITRE 34

Services spécifiques à MS Windows

Ce chapitre documente les modules qui ne sont disponibles que sous MS Windows.

34.1 msvcrt --- Useful routines from the MS VC++ runtime

These functions provide access to some useful capabilities on Windows platforms. Some higher-level modules use these functions to build the Windows implementations of their services. For example, the `getpass` module uses this in the implementation of the `getpass()` function.

Further documentation on these functions can be found in the Platform API documentation.

The module implements both the normal and wide char variants of the console I/O api. The normal API deals only with ASCII characters and is of limited use for internationalized applications. The wide char API should be used where ever possible.

Modifié dans la version 3.3 : Les opérations de ce module lèvent désormais `OSError` au lieu de `IOError`.

34.1.1 Opérations sur les fichiers

`msvcrt.`**`locking`**(*fd*, *mode*, *nbytes*)

Lock part of a file based on file descriptor *fd* from the C runtime. Raises `OSError` on failure. The locked region of the file extends from the current file position for *nbytes* bytes, and may continue beyond the end of the file. *mode* must be one of the `LK_*` constants listed below. Multiple regions in a file may be locked at the same time, but may not overlap. Adjacent regions are not merged ; they must be unlocked individually.

Raises an *auditing event* `msvcrt.locking` with arguments `fd`, `mode`, `nbytes`.

`msvcrt.`**`LK_LOCK`**

`msvcrt.`**`LK_RLCK`**

Locks the specified bytes. If the bytes cannot be locked, the program immediately tries again after 1 second. If, after 10 attempts, the bytes cannot be locked, `OSError` is raised.

`msvcrt.`**`LK_NBLCK`**

`msvcrt.`**`LK_NBRLCK`**

Verrouille les octets spécifiés. Si les octets ne peuvent pas être verrouillés, `OSError` est levée.

msvcrt.**LK_UNLCK**

Déverrouille les octets spécifiés, qui doivent avoir été précédemment verrouillés.

msvcrt.**setmode**(*fd*, *flags*)

Set the line-end translation mode for the file descriptor *fd*. To set it to text mode, *flags* should be `os.O_TEXT`; for binary, it should be `os.O_BINARY`.

msvcrt.**open_osfhandle**(*handle*, *flags*)

Create a C runtime file descriptor from the file handle *handle*. The *flags* parameter should be a bitwise OR of `os.O_APPEND`, `os.O_RDONLY`, `os.O_TEXT` and `os.O_NOINHERIT`. The returned file descriptor may be used as a parameter to `os.fdopen()` to create a file object.

The file descriptor is inheritable by default. Pass `os.O_NOINHERIT` flag to make it non inheritable.

Raises an *auditing event* `msvcrt.open_osfhandle` with arguments `handle`, `flags`.

msvcrt.**get_osfhandle**(*fd*)

Return the file handle for the file descriptor *fd*. Raises `OSError` if *fd* is not recognized.

Raises an *auditing event* `msvcrt.get_osfhandle` with argument `fd`.

34.1.2 Entrées-sorties sur un terminal

msvcrt.**kbhit**()

Returns a nonzero value if a keypress is waiting to be read. Otherwise, return 0.

msvcrt.**getch**()

Read a keypress and return the resulting character as a byte string. Nothing is echoed to the console. This call will block if a keypress is not already available, but will not wait for `Enter` to be pressed. If the pressed key was a special function key, this will return `'\000'` or `'\xe0'`; the next call will return the keycode. The `Control-C` keypress cannot be read with this function.

msvcrt.**getwch**()

Variante de `getch()` qui lit un caractère large et renvoie une valeur Unicode.

msvcrt.**getche**()

Similar to `getch()`, but the keypress will be echoed if it represents a printable character.

msvcrt.**getwche**()

Variante de `getche()` qui lit un caractère large et renvoie une valeur Unicode.

msvcrt.**putch**(*char*)

Print the byte string *char* to the console without buffering.

msvcrt.**putwch**(*unicode_char*)

Variante de `putch()` qui accepte une valeur Unicode et l'écrit comme caractère large.

msvcrt.**ungetch**(*char*)

Cause the byte string *char* to be "pushed back" into the console buffer; it will be the next character read by `getch()` or `getche()`.

msvcrt.**ungetwch**(*unicode_char*)

Variante de `ungetch()` qui accepte une valeur Unicode et l'écrit comme caractère large.

34.1.3 Autres fonctions

msvcrt.**heapmin**()

Force the `malloc()` heap to clean itself up and return unused blocks to the operating system. On failure, this raises `OSError`.

msvcrt.**set_error_mode**(*mode*)

Changes the location where the C runtime writes an error message for an error that might end the program. *mode* must be one of the `OUT_*` constants listed below or `REPORT_ERRMODE`. Returns the old setting or -1 if an error occurs. Only available in debug build of Python.

msvcrt.**OUT_TO_DEFAULT**

Error sink is determined by the app's type. Only available in debug build of Python.

msvcrt.**OUT_TO_STDERR**

Error sink is a standard error. Only available in debug build of Python.

msvcrt.**OUT_TO_MSGBOX**

Error sink is a message box. Only available in debug build of Python.

msvcrt.**REPORT_ERRMODE**

Report the current error mode value. Only available in debug build of Python.

msvcrt.**CrtSetReportMode**(*type*, *mode*)

Specifies the destination or destinations for a specific report type generated by `_CrtDbgReport()` in the MS VC++ runtime. *type* must be one of the `CRT_*` constants listed below. *mode* must be one of the `CRTDBG_*` constants listed below. Only available in debug build of Python.

msvcrt.**CrtSetReportFile**(*type*, *file*)

After you use `CrtSetReportMode()` to specify `CRTDBG_MODE_FILE`, you can specify the file handle to receive the message text. *type* must be one of the `CRT_*` constants listed below. *file* should be the file handle your want specified. Only available in debug build of Python.

msvcrt.**CRT_WARN**

Warnings, messages, and information that doesn't need immediate attention.

msvcrt.**CRT_ERROR**

Errors, unrecoverable problems, and issues that require immediate attention.

msvcrt.**CRT_ASSERT**

Assertion failures.

msvcrt.**CRTDBG_MODE_DEBUG**

Writes the message to the debugger's output window.

msvcrt.**CRTDBG_MODE_FILE**

Writes the message to a user-supplied file handle. `CrtSetReportFile()` should be called to define the specific file or stream to use as the destination.

msvcrt.**CRTDBG_MODE_WNDW**

Creates a message box to display the message along with the `Abort`, `Retry`, and `Ignore` buttons.

msvcrt.**CRTDBG_REPORT_MODE**

Returns current *mode* for the specified *type*.

msvcrt.**CRT_ASSEMBLY_VERSION**

The CRT Assembly version, from the `crtassem.h` header file.

msvcrt.**VC_ASSEMBLY_PUBLICKEYTOKEN**

The VC Assembly public key token, from the `crtassem.h` header file.

msvcrt.**LIBRARIES_ASSEMBLY_NAME_PREFIX**

The Libraries Assembly name prefix, from the `crtassem.h` header file.

34.2 winreg --- Windows registry access

These functions expose the Windows registry API to Python. Instead of using an integer as the registry handle, a *handle object* is used to ensure that the handles are closed correctly, even if the programmer neglects to explicitly close them.

Modifié dans la version 3.3 : Several functions in this module used to raise a `WindowsError`, which is now an alias of `OSError`.

34.2.1 Fonctions

This module offers the following functions :

`winreg.CloseKey`(*hkey*)

Closes a previously opened registry key. The *hkey* argument specifies a previously opened key.

> **Note**
>
> If *hkey* is not closed using this method (or via `hkey.Close()`), it is closed when the *hkey* object is destroyed by Python.

`winreg.ConnectRegistry`(*computer_name*, *key*)

Establishes a connection to a predefined registry handle on another computer, and returns a *handle object*.

computer_name is the name of the remote computer, of the form `r"\\computername"`. If `None`, the local computer is used.

key is the predefined handle to connect to.

The return value is the handle of the opened key. If the function fails, an `OSError` exception is raised.

Raises an *auditing event* `winreg.ConnectRegistry` with arguments `computer_name`, `key`.

Modifié dans la version 3.3 : See *above*.

`winreg.CreateKey`(*key*, *sub_key*)

Creates or opens the specified key, returning a *handle object*.

key is an already open key, or one of the predefined *HKEY_* constants*.

sub_key is a string that names the key this method opens or creates.

If *key* is one of the predefined keys, *sub_key* may be `None`. In that case, the handle returned is the same key handle passed in to the function.

If the key already exists, this function opens the existing key.

The return value is the handle of the opened key. If the function fails, an `OSError` exception is raised.

Raises an *auditing event* `winreg.CreateKey` with arguments `key`, `sub_key`, `access`.

Raises an *auditing event* `winreg.OpenKey/result` with argument `key`.

Modifié dans la version 3.3 : See *above*.

`winreg.CreateKeyEx`(*key*, *sub_key*, *reserved=0*, *access=KEY_WRITE*)

Creates or opens the specified key, returning a *handle object*.

key is an already open key, or one of the predefined *HKEY_* constants*.

sub_key is a string that names the key this method opens or creates.

reserved is a reserved integer, and must be zero. The default is zero.

access is an integer that specifies an access mask that describes the desired security access for the key. Default is `KEY_WRITE`. See *Access Rights* for other allowed values.

If *key* is one of the predefined keys, *sub_key* may be `None`. In that case, the handle returned is the same key handle passed in to the function.

If the key already exists, this function opens the existing key.

The return value is the handle of the opened key. If the function fails, an `OSError` exception is raised.

Raises an *auditing event* `winreg.CreateKey` with arguments `key`, `sub_key`, `access`.

Raises an *auditing event* `winreg.OpenKey/result` with argument `key`.

Ajouté dans la version 3.2.

Modifié dans la version 3.3 : See *above*.

`winreg.`**`DeleteKey`**(*key*, *sub_key*)

Deletes the specified key.

key is an already open key, or one of the predefined *HKEY_* constants*.

sub_key is a string that must be a subkey of the key identified by the *key* parameter. This value must not be `None`, and the key may not have subkeys.

This method can not delete keys with subkeys.

If the method succeeds, the entire key, including all of its values, is removed. If the method fails, an `OSError` exception is raised.

Raises an *auditing event* `winreg.DeleteKey` with arguments `key`, `sub_key`, `access`.

Modifié dans la version 3.3 : See *above*.

`winreg.`**`DeleteKeyEx`**(*key*, *sub_key*, *access=KEY_WOW64_64KEY*, *reserved=0*)

Deletes the specified key.

key is an already open key, or one of the predefined *HKEY_* constants*.

sub_key is a string that must be a subkey of the key identified by the *key* parameter. This value must not be `None`, and the key may not have subkeys.

reserved is a reserved integer, and must be zero. The default is zero.

access is an integer that specifies an access mask that describes the desired security access for the key. Default is `KEY_WOW64_64KEY`. On 32-bit Windows, the WOW64 constants are ignored. See *Access Rights* for other allowed values.

This method can not delete keys with subkeys.

If the method succeeds, the entire key, including all of its values, is removed. If the method fails, an `OSError` exception is raised.

On unsupported Windows versions, `NotImplementedError` is raised.

Raises an *auditing event* `winreg.DeleteKey` with arguments `key`, `sub_key`, `access`.

Ajouté dans la version 3.2.

Modifié dans la version 3.3 : See *above*.

`winreg.`**`DeleteValue`**(*key*, *value*)

Removes a named value from a registry key.

key is an already open key, or one of the predefined *HKEY_* constants*.

value is a string that identifies the value to remove.

Raises an *auditing event* `winreg.DeleteValue` with arguments `key`, `value`.

`winreg.`**`EnumKey`**(*key*, *index*)

Enumerates subkeys of an open registry key, returning a string.

key is an already open key, or one of the predefined *HKEY_* constants*.

index is an integer that identifies the index of the key to retrieve.

The function retrieves the name of one subkey each time it is called. It is typically called repeatedly until an `OSError` exception is raised, indicating, no more values are available.

Raises an *auditing event* `winreg.EnumKey` with arguments `key`, `index`.

Modifié dans la version 3.3 : See *above*.

`winreg.`**`EnumValue`**(*key*, *index*)

Enumerates values of an open registry key, returning a tuple.

key is an already open key, or one of the predefined *HKEY_* constants*.

index is an integer that identifies the index of the value to retrieve.

The function retrieves the name of one subkey each time it is called. It is typically called repeatedly, until an `OSError` exception is raised, indicating no more values.

The result is a tuple of 3 items :

Index	Signification
0	A string that identifies the value name
1	An object that holds the value data, and whose type depends on the underlying registry type
2	An integer that identifies the type of the value data (see table in docs for `SetValueEx()`)

Raises an *auditing event* `winreg.EnumValue` with arguments `key`, `index`.

Modifié dans la version 3.3 : See *above*.

`winreg.`**`ExpandEnvironmentStrings`**(*str*)

Expands environment variable placeholders `%NAME%` in strings like `REG_EXPAND_SZ` :

```
>>> ExpandEnvironmentStrings('%windir%')
'C:\\Windows'
```

Raises an *auditing event* `winreg.ExpandEnvironmentStrings` with argument `str`.

`winreg.`**`FlushKey`**(*key*)

Writes all the attributes of a key to the registry.

key is an already open key, or one of the predefined *HKEY_* constants*.

It is not necessary to call `FlushKey()` to change a key. Registry changes are flushed to disk by the registry using its lazy flusher. Registry changes are also flushed to disk at system shutdown. Unlike `CloseKey()`, the `FlushKey()` method returns only when all the data has been written to the registry. An application should only call `FlushKey()` if it requires absolute certainty that registry changes are on disk.

> **Note**
>
> If you don't know whether a `FlushKey()` call is required, it probably isn't.

`winreg.`**`LoadKey`**(*key*, *sub_key*, *file_name*)

Creates a subkey under the specified key and stores registration information from a specified file into that subkey.

key is a handle returned by `ConnectRegistry()` or one of the constants `HKEY_USERS` or `HKEY_LOCAL_MACHINE`.

sub_key is a string that identifies the subkey to load.

file_name is the name of the file to load registry data from. This file must have been created with the `SaveKey()` function. Under the file allocation table (FAT) file system, the filename may not have an extension.

A call to `LoadKey()` fails if the calling process does not have the `SE_RESTORE_PRIVILEGE` privilege. Note that privileges are different from permissions -- see the RegLoadKey documentation for more details.

If *key* is a handle returned by `ConnectRegistry()`, then the path specified in *file_name* is relative to the remote computer.

Raises an *auditing event* `winreg.LoadKey` with arguments `key`, `sub_key`, `file_name`.

`winreg.`**`OpenKey`**(*key*, *sub_key*, *reserved=0*, *access=KEY_READ*)

`winreg.`**`OpenKeyEx`**(*key*, *sub_key*, *reserved=0*, *access=KEY_READ*)

Opens the specified key, returning a *handle object*.

key is an already open key, or one of the predefined *HKEY_* constants*.

sub_key is a string that identifies the sub_key to open.

reserved is a reserved integer, and must be zero. The default is zero.

access is an integer that specifies an access mask that describes the desired security access for the key. Default is `KEY_READ`. See *Access Rights* for other allowed values.

The result is a new handle to the specified key.

If the function fails, `OSError` is raised.

Raises an *auditing event* `winreg.OpenKey` with arguments `key`, `sub_key`, `access`.

Raises an *auditing event* `winreg.OpenKey/result` with argument `key`.

Modifié dans la version 3.2 : Allow the use of named arguments.

Modifié dans la version 3.3 : See *above*.

`winreg.`**`QueryInfoKey`**(*key*)

Returns information about a key, as a tuple.

key is an already open key, or one of the predefined *HKEY_* constants*.

The result is a tuple of 3 items :

Index	Signification
0	An integer giving the number of sub keys this key has.
1	An integer giving the number of values this key has.
2	An integer giving when the key was last modified (if available) as 100's of nanoseconds since Jan 1, 1601.

Raises an *auditing event* `winreg.QueryInfoKey` with argument `key`.

`winreg.`**`QueryValue`**(*key*, *sub_key*)

Retrieves the unnamed value for a key, as a string.

key is an already open key, or one of the predefined *HKEY_* constants*.

sub_key is a string that holds the name of the subkey with which the value is associated. If this parameter is `None` or empty, the function retrieves the value set by the `SetValue()` method for the key identified by *key*.

Values in the registry have name, type, and data components. This method retrieves the data for a key's first value that has a `NULL` name. But the underlying API call doesn't return the type, so always use `QueryValueEx()` if possible.

Raises an *auditing event* `winreg.QueryValue` with arguments `key`, `sub_key`, `value_name`.

winreg.**QueryValueEx**(*key*, *value_name*)

Retrieves the type and data for a specified value name associated with an open registry key.

key is an already open key, or one of the predefined *HKEY_* constants*.

value_name is a string indicating the value to query.

The result is a tuple of 2 items :

Index	Signification
0	The value of the registry item.
1	An integer giving the registry type for this value (see table in docs for `SetValueEx()`)

Raises an *auditing event* `winreg.QueryValue` with arguments `key`, `sub_key`, `value_name`.

winreg.**SaveKey**(*key*, *file_name*)

Saves the specified key, and all its subkeys to the specified file.

key is an already open key, or one of the predefined *HKEY_* constants*.

file_name is the name of the file to save registry data to. This file cannot already exist. If this filename includes an extension, it cannot be used on file allocation table (FAT) file systems by the `LoadKey()` method.

If *key* represents a key on a remote computer, the path described by *file_name* is relative to the remote computer. The caller of this method must possess the **SeBackupPrivilege** security privilege. Note that privileges are different than permissions -- see the Conflicts Between User Rights and Permissions documentation for more details.

This function passes `NULL` for *security_attributes* to the API.

Raises an *auditing event* `winreg.SaveKey` with arguments `key`, `file_name`.

winreg.**SetValue**(*key*, *sub_key*, *type*, *value*)

Associates a value with a specified key.

key is an already open key, or one of the predefined *HKEY_* constants*.

sub_key is a string that names the subkey with which the value is associated.

type is an integer that specifies the type of the data. Currently this must be `REG_SZ`, meaning only strings are supported. Use the `SetValueEx()` function for support for other data types.

value is a string that specifies the new value.

If the key specified by the *sub_key* parameter does not exist, the SetValue function creates it.

Value lengths are limited by available memory. Long values (more than 2048 bytes) should be stored as files with the filenames stored in the configuration registry. This helps the registry perform efficiently.

The key identified by the *key* parameter must have been opened with `KEY_SET_VALUE` access.

Raises an *auditing event* `winreg.SetValue` with arguments `key`, `sub_key`, `type`, `value`.

winreg.**SetValueEx**(*key*, *value_name*, *reserved*, *type*, *value*)

Stores data in the value field of an open registry key.

key is an already open key, or one of the predefined *HKEY_* constants*.

value_name is a string that names the subkey with which the value is associated.

reserved can be anything -- zero is always passed to the API.

type is an integer that specifies the type of the data. See *Value Types* for the available types.

value is a string that specifies the new value.

This method can also set additional value and type information for the specified key. The key identified by the key parameter must have been opened with `KEY_SET_VALUE` access.

To open the key, use the `CreateKey()` or `OpenKey()` methods.

Value lengths are limited by available memory. Long values (more than 2048 bytes) should be stored as files with the filenames stored in the configuration registry. This helps the registry perform efficiently.

Raises an *auditing event* `winreg.SetValue` with arguments `key`, `sub_key`, `type`, `value`.

`winreg.`**`DisableReflectionKey`**(*key*)

Disables registry reflection for 32-bit processes running on a 64-bit operating system.

key is an already open key, or one of the predefined *HKEY_* constants*.

Will generally raise `NotImplementedError` if executed on a 32-bit operating system.

If the key is not on the reflection list, the function succeeds but has no effect. Disabling reflection for a key does not affect reflection of any subkeys.

Raises an *auditing event* `winreg.DisableReflectionKey` with argument `key`.

`winreg.`**`EnableReflectionKey`**(*key*)

Restores registry reflection for the specified disabled key.

key is an already open key, or one of the predefined *HKEY_* constants*.

Will generally raise `NotImplementedError` if executed on a 32-bit operating system.

Restoring reflection for a key does not affect reflection of any subkeys.

Raises an *auditing event* `winreg.EnableReflectionKey` with argument `key`.

`winreg.`**`QueryReflectionKey`**(*key*)

Determines the reflection state for the specified key.

key is an already open key, or one of the predefined *HKEY_* constants*.

Returns `True` if reflection is disabled.

Will generally raise `NotImplementedError` if executed on a 32-bit operating system.

Raises an *auditing event* `winreg.QueryReflectionKey` with argument `key`.

34.2.2 Constantes

The following constants are defined for use in many `winreg` functions.

HKEY_* Constants

`winreg.`**`HKEY_CLASSES_ROOT`**

Registry entries subordinate to this key define types (or classes) of documents and the properties associated with those types. Shell and COM applications use the information stored under this key.

`winreg.`**`HKEY_CURRENT_USER`**

Registry entries subordinate to this key define the preferences of the current user. These preferences include the settings of environment variables, data about program groups, colors, printers, network connections, and application preferences.

`winreg.`**`HKEY_LOCAL_MACHINE`**

Registry entries subordinate to this key define the physical state of the computer, including data about the bus type, system memory, and installed hardware and software.

`winreg.`**`HKEY_USERS`**

Registry entries subordinate to this key define the default user configuration for new users on the local computer and the user configuration for the current user.

`winreg.`**`HKEY_PERFORMANCE_DATA`**

Registry entries subordinate to this key allow you to access performance data. The data is not actually stored in the registry; the registry functions cause the system to collect the data from its source.

winreg.**HKEY_CURRENT_CONFIG**

Contains information about the current hardware profile of the local computer system.

winreg.**HKEY_DYN_DATA**

This key is not used in versions of Windows after 98.

Access Rights

For more information, see Registry Key Security and Access.

winreg.**KEY_ALL_ACCESS**

Combines the STANDARD_RIGHTS_REQUIRED, *KEY_QUERY_VALUE*, *KEY_SET_VALUE*, *KEY_CREATE_SUB_KEY*, *KEY_ENUMERATE_SUB_KEYS*, *KEY_NOTIFY*, and *KEY_CREATE_LINK* access rights.

winreg.**KEY_WRITE**

Combines the STANDARD_RIGHTS_WRITE, *KEY_SET_VALUE*, and *KEY_CREATE_SUB_KEY* access rights.

winreg.**KEY_READ**

Combines the STANDARD_RIGHTS_READ, *KEY_QUERY_VALUE*, *KEY_ENUMERATE_SUB_KEYS*, and *KEY_NOTIFY* values.

winreg.**KEY_EXECUTE**

Equivalent to *KEY_READ*.

winreg.**KEY_QUERY_VALUE**

Required to query the values of a registry key.

winreg.**KEY_SET_VALUE**

Required to create, delete, or set a registry value.

winreg.**KEY_CREATE_SUB_KEY**

Required to create a subkey of a registry key.

winreg.**KEY_ENUMERATE_SUB_KEYS**

Required to enumerate the subkeys of a registry key.

winreg.**KEY_NOTIFY**

Required to request change notifications for a registry key or for subkeys of a registry key.

winreg.**KEY_CREATE_LINK**

Reserved for system use.

64-bit Specific

For more information, see Accessing an Alternate Registry View.

winreg.**KEY_WOW64_64KEY**

Indicates that an application on 64-bit Windows should operate on the 64-bit registry view. On 32-bit Windows, this constant is ignored.

winreg.**KEY_WOW64_32KEY**

Indicates that an application on 64-bit Windows should operate on the 32-bit registry view. On 32-bit Windows, this constant is ignored.

Value Types

For more information, see Registry Value Types.

winreg.**REG_BINARY**

Binary data in any form.

winreg.**REG_DWORD**

32-bit number.

winreg.**REG_DWORD_LITTLE_ENDIAN**

A 32-bit number in little-endian format. Equivalent to *REG_DWORD*.

winreg.**REG_DWORD_BIG_ENDIAN**

A 32-bit number in big-endian format.

winreg.**REG_EXPAND_SZ**

Null-terminated string containing references to environment variables (`%PATH%`).

winreg.**REG_LINK**

A Unicode symbolic link.

winreg.**REG_MULTI_SZ**

A sequence of null-terminated strings, terminated by two null characters. (Python handles this termination automatically.)

winreg.**REG_NONE**

No defined value type.

winreg.**REG_QWORD**

A 64-bit number.

Ajouté dans la version 3.6.

winreg.**REG_QWORD_LITTLE_ENDIAN**

A 64-bit number in little-endian format. Equivalent to *REG_QWORD*.

Ajouté dans la version 3.6.

winreg.**REG_RESOURCE_LIST**

A device-driver resource list.

winreg.**REG_FULL_RESOURCE_DESCRIPTOR**

A hardware setting.

winreg.**REG_RESOURCE_REQUIREMENTS_LIST**

A hardware resource list.

winreg.**REG_SZ**

A null-terminated string.

34.2.3 Registry Handle Objects

This object wraps a Windows HKEY object, automatically closing it when the object is destroyed. To guarantee cleanup, you can call either the *Close()* method on the object, or the *CloseKey()* function.

All registry functions in this module return one of these objects.

All registry functions in this module which accept a handle object also accept an integer, however, use of the handle object is encouraged.

Handle objects provide semantics for `__bool__()` -- thus

```
if handle:
    print("Yes")
```

will print `Yes` if the handle is currently valid (has not been closed or detached).

The object also support comparison semantics, so handle objects will compare true if they both reference the same underlying Windows handle value.

Handle objects can be converted to an integer (e.g., using the built-in `int()` function), in which case the underlying Windows handle value is returned. You can also use the `Detach()` method to return the integer handle, and also disconnect the Windows handle from the handle object.

`PyHKEY.Close()`

Closes the underlying Windows handle.

If the handle is already closed, no error is raised.

`PyHKEY.Detach()`

Detaches the Windows handle from the handle object.

The result is an integer that holds the value of the handle before it is detached. If the handle is already detached or closed, this will return zero.

After calling this function, the handle is effectively invalidated, but the handle is not closed. You would call this function when you need the underlying Win32 handle to exist beyond the lifetime of the handle object.

Raises an *auditing event* `winreg.PyHKEY.Detach` with argument `key`.

`PyHKEY.__enter__()`

`PyHKEY.__exit__(*exc_info)`

The HKEY object implements `__enter__()` and `__exit__()` and thus supports the context protocol for the `with` statement :

```
with OpenKey(HKEY_LOCAL_MACHINE, "foo") as key:
    ...  # work with key
```

will automatically close *key* when control leaves the `with` block.

34.3 `winsound` --- Sound-playing interface for Windows

The `winsound` module provides access to the basic sound-playing machinery provided by Windows platforms. It includes functions and several constants.

`winsound.Beep(frequency, duration)`

Beep the PC's speaker. The *frequency* parameter specifies frequency, in hertz, of the sound, and must be in the range 37 through 32,767. The *duration* parameter specifies the number of milliseconds the sound should last. If the system is not able to beep the speaker, `RuntimeError` is raised.

`winsound.PlaySound(sound, flags)`

Call the underlying `PlaySound()` function from the Platform API. The *sound* parameter may be a filename, a system sound alias, audio data as a *bytes-like object*, or `None`. Its interpretation depends on the value of *flags*, which can be a bitwise ORed combination of the constants described below. If the *sound* parameter is `None`, any currently playing waveform sound is stopped. If the system indicates an error, `RuntimeError` is raised.

`winsound.MessageBeep(type=MB_OK)`

Call the underlying `MessageBeep()` function from the Platform API. This plays a sound as specified in the registry. The *type* argument specifies which sound to play; possible values are `-1`, `MB_ICONASTERISK`, `MB_ICONEXCLAMATION`, `MB_ICONHAND`, `MB_ICONQUESTION`, and `MB_OK`, all described below. The value `-1` produces a "simple beep"; this is the final fallback if a sound cannot be played otherwise. If the system indicates an error, `RuntimeError` is raised.

`winsound.SND_FILENAME`

The *sound* parameter is the name of a WAV file. Do not use with `SND_ALIAS`.

`winsound.SND_ALIAS`

The *sound* parameter is a sound association name from the registry. If the registry contains no such name, play the system default sound unless `SND_NODEFAULT` is also specified. If no default sound is registered, raise `RuntimeError`. Do not use with `SND_FILENAME`.

All Win32 systems support at least the following ; most systems support many more :

PlaySound() name	Corresponding Control Panel Sound name
'SystemAsterisk'	Asterisk
'SystemExclamation'	Exclamation
'SystemExit'	Exit Windows
'SystemHand'	Critical Stop
'SystemQuestion'	Question

Par exemple :

```
import winsound
# Play Windows exit sound.
winsound.PlaySound("SystemExit", winsound.SND_ALIAS)

# Probably play Windows default sound, if any is registered (because
# "*" probably isn't the registered name of any sound).
winsound.PlaySound("*", winsound.SND_ALIAS)
```

winsound.**SND_LOOP**

Play the sound repeatedly. The *SND_ASYNC* flag must also be used to avoid blocking. Cannot be used with *SND_MEMORY*.

winsound.**SND_MEMORY**

The *sound* parameter to *PlaySound()* is a memory image of a WAV file, as a *bytes-like object*.

> **Note**
>
> This module does not support playing from a memory image asynchronously, so a combination of this flag and *SND_ASYNC* will raise *RuntimeError*.

winsound.**SND_PURGE**

Stop playing all instances of the specified sound.

> **Note**
>
> This flag is not supported on modern Windows platforms.

winsound.**SND_ASYNC**

Return immediately, allowing sounds to play asynchronously.

winsound.**SND_NODEFAULT**

If the specified sound cannot be found, do not play the system default sound.

winsound.**SND_NOSTOP**

Do not interrupt sounds currently playing.

winsound.**SND_NOWAIT**

Return immediately if the sound driver is busy.

> **Note**
>
> This flag is not supported on modern Windows platforms.

`winsound.`**`MB_ICONASTERISK`**

Play the `SystemDefault` sound.

`winsound.`**`MB_ICONEXCLAMATION`**

Play the `SystemExclamation` sound.

`winsound.`**`MB_ICONHAND`**

Play the `SystemHand` sound.

`winsound.`**`MB_ICONQUESTION`**

Play the `SystemQuestion` sound.

`winsound.`**`MB_OK`**

Play the `SystemDefault` sound.

CHAPITRE 35

Services spécifiques à Unix

Les modules décrits dans ce chapitre fournissent des interfaces aux fonctionnalités propres au système d'exploitation Unix ou, dans certains cas, à certaines de ses variantes, en voici un aperçu :

35.1 `posix` --- The most common POSIX system calls

Ce module permet d'accéder aux fonctionnalités du système d'exploitation normalisés par le Standard C et le Standard POSIX (une interface Unix habilement déguisée).

Availability : Unix.

Ne pas importer ce module directement. À la place, importer le module `os`, qui fournit une version *portable* de cette interface. Sous Unix, le module `os` fournit un sur-ensemble de l'interface `posix`. Sous les systèmes d'exploitation non Unix le module `posix` n'est pas disponible, mais un sous ensemble est toujours disponible via l'interface `os`. Une fois que `os` est importé, il n'y a aucune perte de performance à l'utiliser à la place de `posix`. De plus, `os` fournit des fonctionnalités supplémentaires, telles que l'appel automatique de `putenv()` lorsqu'une entrée dans `os.environ` est modifiée.

Les erreurs sont signalées comme des exceptions ; les exceptions habituelles sont données pour les erreurs de type, tandis que les erreurs signalées par les appels système lèvent une erreur `OSError`.

35.1.1 Prise en charge de gros fichiers

Several operating systems (including AIX and Solaris) provide support for files that are larger than 2 GiB from a C programming model where `int` and `long` are 32-bit values. This is typically accomplished by defining the relevant size and offset types as 64-bit values. Such files are sometimes referred to as *large files*.

Large file support is enabled in Python when the size of an `off_t` is larger than a `long` and the `long long` is at least as large as an `off_t`. It may be necessary to configure and compile Python with certain compiler flags to enable this mode. For example, with Solaris 2.6 and 2.7 you need to do something like :

```
CFLAGS="`getconf LFS_CFLAGS`" OPT="-g -O2 $CFLAGS" \
        ./configure
```

Sur les systèmes Linux capable de supporter les fichiers volumineux, cela pourrait fonctionner :

```
CFLAGS='-D_LARGEFILE64_SOURCE -D_FILE_OFFSET_BITS=64' OPT="-g -O2 $CFLAGS" \
        ./configure
```

35.1.2 Contenu du Module

En plus des nombreuses fonctions décrites dans la documentation du module *os*, *posix* possède les éléments suivants :

`posix.environ`

Un dictionnaire représentant les variables d'environnement au moment où l'interpréteur à été lancé. Les clés et les valeurs sont des `bytes` sous Unix et des `str` sous Windows. Par exemple, `environ[b'HOME']` (`environ['HOME']` dans Windows) est le chemin de votre dossier d'accueil, équivalent à `getenv("HOME")` en C.

Modifier ce dictionnaire n'affecte pas les variables d'environnements fournis par *execv()*, *popen()* ou *system()* ; Si vous avez besoin de changer l'environnement, passer le paramètre `environ` à *execve()* ou ajouter les assignations de variables et les *export* à la commande à exécuter via *system()* ou *popen()*.

Modifié dans la version 3.2 : Sous Unix, les clés et les valeurs sont des octets.

> **Note**
>
> Le module *os* fournit une implémentation alternative à `environ` qui met à jour l'environnement en cas de modification. Notez également que la mise à jour de *os.environ* rendra ce dictionnaire obsolète. Il est recommandé d'utiliser le module *os* au lieu du module *posix* dans ce cas-ci.

35.2 `pwd` --- The password database

This module provides access to the Unix user account and password database. It is available on all Unix versions.

Availability : Unix, not WASI, not iOS.

Password database entries are reported as a tuple-like object, whose attributes correspond to the members of the `passwd` structure (Attribute field below, see `<pwd.h>`) :

Index	Attribut	Signification
0	`pw_name`	Nom d'utilisateur
1	`pw_passwd`	Optional encrypted password
2	`pw_uid`	Numerical user ID
3	`pw_gid`	Numerical group ID
4	`pw_gecos`	User name or comment field
5	`pw_dir`	Répertoire d'accueil de l'utilisateur
6	`pw_shell`	User command interpreter

The uid and gid items are integers, all others are strings. *KeyError* is raised if the entry asked for cannot be found.

> **Note**
>
> In traditional Unix the field `pw_passwd` usually contains a password encrypted with a DES derived algorithm. However most modern unices use a so-called *shadow password* system. On those unices the *pw_passwd* field only contains an asterisk (`'*'`) or the letter `'x'` where the encrypted password is stored in a file `/etc/shadow` which is not world readable. Whether the *pw_passwd* field contains anything useful is system-dependent.

It defines the following items :

`pwd.getpwuid(uid)`

Return the password database entry for the given numeric user ID.

`pwd.getpwnam(name)`

Return the password database entry for the given user name.

`pwd.getpwall()`

Return a list of all available password database entries, in arbitrary order.

Voir aussi

Module *grp*
Interface pour la base de données des groupes, similaire à celle-ci.

35.3 `grp` --- The group database

This module provides access to the Unix group database. It is available on all Unix versions.

Availability : Unix, not WASI, not Android, not iOS.

Group database entries are reported as a tuple-like object, whose attributes correspond to the members of the `group` structure (Attribute field below, see `<grp.h>`) :

Index	Attribut	Signification
0	gr_name	the name of the group
1	gr_passwd	the (encrypted) group password ; often empty
2	gr_gid	the numerical group ID
3	gr_mem	all the group member's user names

The gid is an integer, name and password are strings, and the member list is a list of strings. (Note that most users are not explicitly listed as members of the group they are in according to the password database. Check both databases to get complete membership information. Also note that a `gr_name` that starts with a + or − is likely to be a YP/NIS reference and may not be accessible via `getgrnam()` or `getgrgid()`.)

It defines the following items :

`grp.getgrgid(id)`

Return the group database entry for the given numeric group ID. `KeyError` is raised if the entry asked for cannot be found.

Modifié dans la version 3.10 : `TypeError` is raised for non-integer arguments like floats or strings.

`grp.getgrnam(name)`

Return the group database entry for the given group name. `KeyError` is raised if the entry asked for cannot be found.

`grp.getgrall()`

Return a list of all available group entries, in arbitrary order.

Voir aussi

Module *pwd*
An interface to the user database, similar to this.

35.4 termios --- POSIX style tty control

Ce module fournit une interface aux appels POSIX pour le contrôle des entrées-sorties d'un terminal. Pour une description complète de ces appels, voir la page du manuel UNIX *termios(3)*. Il n'est disponible que pour les versions Unix qui gèrent le contrôle des entrées-sorties du terminal à travers des appels POSIX *termios* configurés à l'installation.

Availability : Unix.

Toutes les fonctions de ce module prennent un descripteur de fichier *fd* comme premier argument. Ça peut être un descripteur de fichiers entier, tel que le renvoie `sys.stdin.fileno()`, ou un *file object*, tel que `sys.stdin`.

Ce module définit aussi toutes les constantes nécessaires pour travailler avec les fonctions fournies ici ; elles ont les mêmes noms que leurs équivalents en C. Pour plus d'informations sur l'utilisation de ces terminaux, veuillez vous référer à votre documentation système.

Le module définit les fonctions suivantes :

`termios.`**`tcgetattr`**(*fd*)

Renvoie une liste contenant les attributs du terminal pour le descripteur de fichier *fd*, tel que : `[iflag, oflag, cflag, lflag, ispeed, ospeed, cc]` où *cc* est une liste de caractères spéciaux du terminal (chacun est une chaine de caractère de longueur 1, à l'exception des éléments ayant les indices `VMIN` et `VTIME`, ceux-ci sont alors des entiers quand ces champs sont définis). L'interprétation des options (*flags* en anglais) et des vitesses ainsi que l'indexation dans le tableau *cc* doivent être faites en utilisant les constantes symboliques définies dans le module *termios*.

`termios.`**`tcsetattr`**(*fd*, *when*, *attributes*)

Set the tty attributes for file descriptor *fd* from the *attributes*, which is a list like the one returned by *tcgetattr()*. The *when* argument determines when the attributes are changed :

`termios.`**`TCSANOW`**

Change attributes immediately.

`termios.`**`TCSADRAIN`**

Change attributes after transmitting all queued output.

`termios.`**`TCSAFLUSH`**

Change attributes after transmitting all queued output and discarding all queued input.

`termios.`**`tcsendbreak`**(*fd*, *duration*)

Envoie une pause sur le descripteur de fichier *fd*. Une *duration* à zéro envoie une pause de 0,25 à 0,5 seconde ; une *duration* différente de zéro possède une signification spécifique sur chaque système.

`termios.`**`tcdrain`**(*fd*)

Attend que toutes les sorties écrites dans le descripteur de fichier *fd* soient transmises.

`termios.`**`tcflush`**(*fd*, *queue*)

Vide la queue de données du descripteur de fichier *fd*. Le sélecteur *queue* précise la queue : `TCIFLUSH` pour la queue des entrées, `TCOFLUSH` pour la queue des sorties, ou `TCIOFLUSH` pour les deux queues.

`termios.`**`tcflow`**(*fd*, *action*)

Suspend ou reprend l'entrée ou la sortie du descripteur de fichier *fd*. L'argument *action* peut être `TCOOFF` pour suspendre la sortie, `TCOON` pour relancer la sortie, `TCIOFF` pour suspendre l'entrée, ou `TCION` pour relancer l'entrée.

`termios.`**`tcgetwinsize`**(*fd*)

Return a tuple `(ws_row, ws_col)` containing the tty window size for file descriptor *fd*. Requires `termios.TIOCGWINSZ` or `termios.TIOCGSIZE`.

Ajouté dans la version 3.11.

termios.**tcsetwinsize**(*fd*, *winsize*)

Set the tty window size for file descriptor *fd* from *winsize*, which is a two-item tuple `(ws_row, ws_col)` like the one returned by `tcgetwinsize()`. Requires at least one of the pairs (`termios.TIOCGWINSZ`, `termios.TIOCSWINSZ`); (`termios.TIOCGSIZE`, `termios.TIOCSSIZE`) to be defined.

Ajouté dans la version 3.11.

Voir aussi

Le module `tty`
Fonctions utiles pour les opérations de contrôle communes dans le terminal.

35.4.1 Exemple

Voici une fonction qui demande à l'utilisateur d'entrer un mot de passe sans l'afficher. Remarquez la technique qui consiste à séparer un appel à `tcgetattr()` et une instruction `try… finally` pour s'assurer que les anciens attributs du terminal soient restaurés tels quels quoi qu'il arrive :

```
def getpass(prompt="Password: "):
    import termios, sys
    fd = sys.stdin.fileno()
    old = termios.tcgetattr(fd)
    new = termios.tcgetattr(fd)
    new[3] = new[3] & ~termios.ECHO          # lflags
    try:
        termios.tcsetattr(fd, termios.TCSADRAIN, new)
        passwd = input(prompt)
    finally:
        termios.tcsetattr(fd, termios.TCSADRAIN, old)
    return passwd
```

35.5 `tty` --- Terminal control functions

Code source : Lib/tty.py

Le module `tty` expose des fonctions permettant de mettre le *tty* en mode *cbreak* ou *raw*.

Availability : Unix.

Puisqu'il a besoin du module `termios`, il ne fonctionnera que sur Unix.

Le module `tty` définit les fonctions suivantes :

tty.**cfmakeraw**(*mode*)

Convert the tty attribute list *mode*, which is a list like the one returned by `termios.tcgetattr()`, to that of a tty in raw mode.

Ajouté dans la version 3.12.

tty.**cfmakecbreak**(*mode*)

Convert the tty attribute list *mode*, which is a list like the one returned by `termios.tcgetattr()`, to that of a tty in cbreak mode.

This clears the `ECHO` and `ICANON` local mode flags in *mode* as well as setting the minimum input to 1 byte with no delay.

Ajouté dans la version 3.12.

Modifié dans la version 3.12.2 : The `ICRNL` flag is no longer cleared. This matches Linux and macOS `stty cbreak` behavior and what `setcbreak()` historically did.

`tty.setraw(fd, when=termios.TCSAFLUSH)`

Change the mode of the file descriptor *fd* to raw. If *when* is omitted, it defaults to `termios.TCSAFLUSH`, and is passed to `termios.tcsetattr()`. The return value of `termios.tcgetattr()` is saved before setting *fd* to raw mode ; this value is returned.

Modifié dans la version 3.12 : The return value is now the original tty attributes, instead of `None`.

`tty.setcbreak(fd, when=termios.TCSAFLUSH)`

Change the mode of file descriptor *fd* to cbreak. If *when* is omitted, it defaults to `termios.TCSAFLUSH`, and is passed to `termios.tcsetattr()`. The return value of `termios.tcgetattr()` is saved before setting *fd* to cbreak mode ; this value is returned.

This clears the `ECHO` and `ICANON` local mode flags as well as setting the minimum input to 1 byte with no delay.

Modifié dans la version 3.12 : The return value is now the original tty attributes, instead of `None`.

Modifié dans la version 3.12.2 : The `ICRNL` flag is no longer cleared. This restores the behavior of Python 3.11 and earlier as well as matching what Linux, macOS, & BSDs describe in their `stty(1)` man pages regarding cbreak mode.

Voir aussi

Module `termios`
Interface bas niveau de gestion du terminal.

35.6 `pty` --- Pseudo-terminal utilities

Code source : Lib/pty.py

Le module `pty` expose des fonctions de manipulation de pseudo-terminaux, il permet d'écrire un programme capable de démarrer un autre processus, d'écrire et de lire depuis son terminal.

Availability : Unix.

Pseudo-terminal handling is highly platform dependent. This code is mainly tested on Linux, FreeBSD, and macOS (it is supposed to work on other POSIX platforms but it's not been thoroughly tested).

Le module `pty` expose les fonctions suivantes :

`pty.fork()`

Fork. Connecte le terminal contrôlé par le fils à un pseudo-terminal. La valeur renvoyée est `(pid, fd)`. Notez que le fils obtient `0` comme *pid* et un *fd* non valide. Le parent obtient le *pid* du fils, et *fd* un descripteur de fichier connecté à un terminal contrôlé par le fils (et donc à l'entrée et la sortie standard du fils).

Avertissement

On macOS the use of this function is unsafe when mixed with using higher-level system APIs, and that includes using `urllib.request`.

`pty.openpty()`

Ouvre une nouvelle paire de pseudo-terminaux, en utilisant si possible `os.openpty()`, ou du code émulant la fonctionnalité pour des systèmes *Unix* génériques. Renvoie une paire de descripteurs de fichier `(master, slave)`, pour le maître et pour l'esclave respectivement.

pty.**spawn**(*argv*[, *master_read*[, *stdin_read*]])

Crée un nouveau processus et connecte son terminal aux entrées/sorties standard du processus courant. Cette stratégie est typiquement utilisée pour les programmes qui veulent lire depuis leur propre terminal. Le processus créé utilisant le *pty* est supposé se terminer et, quand il le fera, l'appel de *spawn* terminera.

A loop copies STDIN of the current process to the child and data received from the child to STDOUT of the current process. It is not signaled to the child if STDIN of the current process closes down.

The functions *master_read* and *stdin_read* are passed a file descriptor which they should read from, and they should always return a byte string. In order to force spawn to return before the child process exits an empty byte array should be returned to signal end of file.

L'implémentation par défaut pour les deux fonctions lit et renvoie jusqu'à 1024 octets à chaque appel de la fonction. La fonction de rappel *master_read* reçoit le descripteur de fichier du pseudo-terminal maître pour lire la sortie du processus enfant, et *stdin_read* reçoit le descripteur de fichier 0, pour lire l'entrée standard du processus parent.

Le renvoi d'une chaîne d'octets vide à partir de l'un ou l'autre des rappels est interprété comme une condition de fin de fichier (EOF), et ce rappel ne sera pas appelé après cela. Si *stdin_read* signale EOF, le terminal de contrôle ne peut plus communiquer avec le processus parent OU le processus enfant. À moins que le processus enfant ne quitte sans aucune entrée, *spawn* sera lancé dans une boucle infinie. Si *master_read* indique la fin de fichier, on aura le même comportement (sur Linux au moins).

Return the exit status value from `os.waitpid()` on the child process.

`os.waitstatus_to_exitcode()` can be used to convert the exit status into an exit code.

Raises an *auditing event* `pty.spawn` with argument `argv`.

Modifié dans la version 3.4 : `spawn()` renvoie maintenant la valeur renvoyée par `os.waitpid()` sur le processus fils.

35.6.1 Exemple

Le programme suivant se comporte comme la commande Unix `script(1)`, utilisant un pseudo-terminal pour enregistrer toutes les entrées et sorties d'une session dans un fichier *typescript*.

```
import argparse
import os
import pty
import sys
import time

parser = argparse.ArgumentParser()
parser.add_argument('-a', dest='append', action='store_true')
parser.add_argument('-p', dest='use_python', action='store_true')
parser.add_argument('filename', nargs='?', default='typescript')
options = parser.parse_args()

shell = sys.executable if options.use_python else os.environ.get('SHELL', 'sh')
filename = options.filename
mode = 'ab' if options.append else 'wb'

with open(filename, mode) as script:
    def read(fd):
        data = os.read(fd, 1024)
        script.write(data)
        return data

    print('Script started, file is', filename)
    script.write(('Script started on %s\n' % time.asctime()).encode())
```

(suite sur la page suivante)

(suite de la page précédente)

```
    pty.spawn(shell, read)

    script.write(('Script done on %s\n' % time.asctime()).encode())
    print('Script done, file is', filename)
```

35.7 fcntl --- The fcntl and ioctl system calls

This module performs file and I/O control on file descriptors. It is an interface to the `fcntl()` and `ioctl()` Unix routines. See the *fcntl(2)* and *ioctl(2)* Unix manual pages for full details.

Availability : Unix, not WASI.

All functions in this module take a file descriptor *fd* as their first argument. This can be an integer file descriptor, such as returned by `sys.stdin.fileno()`, or an *io.IOBase* object, such as `sys.stdin` itself, which provides a *fileno()* that returns a genuine file descriptor.

Modifié dans la version 3.3 : Operations in this module used to raise an *IOError* where they now raise an *OSError*.

Modifié dans la version 3.8 : The `fcntl` module now contains `F_ADD_SEALS`, `F_GET_SEALS`, and `F_SEAL_*` constants for sealing of *os.memfd_create()* file descriptors.

Modifié dans la version 3.9 : On macOS, the `fcntl` module exposes the `F_GETPATH` constant, which obtains the path of a file from a file descriptor. On Linux(>=3.15), the `fcntl` module exposes the `F_OFD_GETLK`, `F_OFD_SETLK` and `F_OFD_SETLKW` constants, which are used when working with open file description locks.

Modifié dans la version 3.10 : On Linux >= 2.6.11, the `fcntl` module exposes the `F_GETPIPE_SZ` and `F_SETPIPE_SZ` constants, which allow to check and modify a pipe's size respectively.

Modifié dans la version 3.11 : On FreeBSD, the `fcntl` module exposes the `F_DUP2FD` and `F_DUP2FD_CLOEXEC` constants, which allow to duplicate a file descriptor, the latter setting `FD_CLOEXEC` flag in addition.

Modifié dans la version 3.12 : On Linux >= 4.5, the *fcntl* module exposes the `FICLONE` and `FICLONERANGE` constants, which allow to share some data of one file with another file by reflinking on some filesystems (e.g., btrfs, OCFS2, and XFS). This behavior is commonly referred to as "copy-on-write".

Modifié dans la version 3.13 : On Linux >= 2.6.32, the `fcntl` module exposes the `F_GETOWN_EX`, `F_SETOWN_EX`, `F_OWNER_TID`, `F_OWNER_PID`, `F_OWNER_PGRP` constants, which allow to direct I/O availability signals to a specific thread, process, or process group. On Linux >= 4.13, the `fcntl` module exposes the `F_GET_RW_HINT`, `F_SET_RW_HINT`, `F_GET_FILE_RW_HINT`, `F_SET_FILE_RW_HINT`, and `RWH_WRITE_LIFE_*` constants, which allow to inform the kernel about the relative expected lifetime of writes on a given inode or via a particular open file description. On Linux >= 5.1 and NetBSD, the `fcntl` module exposes the `F_SEAL_FUTURE_WRITE` constant for use with `F_ADD_SEALS` and `F_GET_SEALS` operations. On FreeBSD, the `fcntl` module exposes the `F_READAHEAD`, `F_ISUNIONSTACK`, and `F_KINFO` constants. On macOS and FreeBSD, the `fcntl` module exposes the `F_RDAHEAD` constant. On NetBSD and AIX, the `fcntl` module exposes the `F_CLOSEM` constant. On NetBSD, the `fcntl` module exposes the `F_MAXFD` constant. On macOS and NetBSD, the `fcntl` module exposes the `F_GETNOSIGPIPE` and `F_SETNOSIGPIPE` constant.

Le module définit les fonctions suivantes :

`fcntl.`**`fcntl`**(*fd*, *cmd*, *arg=0*)

Perform the operation *cmd* on file descriptor *fd* (file objects providing a *fileno()* method are accepted as well). The values used for *cmd* are operating system dependent, and are available as constants in the *fcntl* module, using the same names as used in the relevant C header files. The argument *arg* can either be an integer value, or a *bytes* object. With an integer value, the return value of this function is the integer return value of the C `fcntl()` call. When the argument is bytes it represents a binary structure, e.g. created by *struct.pack()*. The binary data is copied to a buffer whose address is passed to the C `fcntl()` call. The return value after a successful call is the contents of the buffer, converted to a *bytes* object. The length of the returned object will be the same as the length of the *arg* argument. This is limited to 1024 bytes. If the information returned

in the buffer by the operating system is larger than 1024 bytes, this is most likely to result in a segmentation violation or a more subtle data corruption.

If the `fcntl()` call fails, an `OSError` is raised.

Raises an *auditing event* `fcntl.fcntl` with arguments `fd`, `cmd`, `arg`.

`fcntl.`**`ioctl`**(*fd*, *request*, *arg=0*, *mutate_flag=True*)

This function is identical to the `fcntl()` function, except that the argument handling is even more complicated.

The *request* parameter is limited to values that can fit in 32-bits. Additional constants of interest for use as the *request* argument can be found in the `termios` module, under the same names as used in the relevant C header files.

The parameter *arg* can be one of an integer, an object supporting the read-only buffer interface (like `bytes`) or an object supporting the read-write buffer interface (like `bytearray`).

In all but the last case, behaviour is as for the `fcntl()` function.

If a mutable buffer is passed, then the behaviour is determined by the value of the *mutate_flag* parameter.

If it is false, the buffer's mutability is ignored and behaviour is as for a read-only buffer, except that the 1024 byte limit mentioned above is avoided -- so long as the buffer you pass is at least as long as what the operating system wants to put there, things should work.

If *mutate_flag* is true (the default), then the buffer is (in effect) passed to the underlying `ioctl()` system call, the latter's return code is passed back to the calling Python, and the buffer's new contents reflect the action of the `ioctl()`. This is a slight simplification, because if the supplied buffer is less than 1024 bytes long it is first copied into a static buffer 1024 bytes long which is then passed to `ioctl()` and copied back into the supplied buffer.

If the `ioctl()` call fails, an `OSError` exception is raised.

Un exemple :

```
>>> import array, fcntl, struct, termios, os
>>> os.getpgrp()
13341
>>> struct.unpack('h', fcntl.ioctl(0, termios.TIOCGPGRP, "  "))[0]
13341
>>> buf = array.array('h', [0])
>>> fcntl.ioctl(0, termios.TIOCGPGRP, buf, 1)
0
>>> buf
array('h', [13341])
```

Raises an *auditing event* `fcntl.ioctl` with arguments `fd`, `request`, `arg`.

`fcntl.`**`flock`**(*fd*, *operation*)

Perform the lock operation *operation* on file descriptor *fd* (file objects providing a `fileno()` method are accepted as well). See the Unix manual `flock(2)` for details. (On some systems, this function is emulated using `fcntl()`.)

If the `flock()` call fails, an `OSError` exception is raised.

Raises an *auditing event* `fcntl.flock` with arguments `fd`, `operation`.

`fcntl.`**`lockf`**(*fd*, *cmd*, *len=0*, *start=0*, *whence=0*)

This is essentially a wrapper around the `fcntl()` locking calls. *fd* is the file descriptor (file objects providing a `fileno()` method are accepted as well) of the file to lock or unlock, and *cmd* is one of the following values :

`fcntl.`**`LOCK_UN`**

Release an existing lock.

`fcntl.LOCK_SH`
Acquire a shared lock.

`fcntl.LOCK_EX`
Acquire an exclusive lock.

`fcntl.LOCK_NB`
Bitwise OR with any of the other three `LOCK_*` constants to make the request non-blocking.

If `LOCK_NB` is used and the lock cannot be acquired, an `OSError` will be raised and the exception will have an *errno* attribute set to `EACCES` or `EAGAIN` (depending on the operating system ; for portability, check for both values). On at least some systems, `LOCK_EX` can only be used if the file descriptor refers to a file opened for writing.

len is the number of bytes to lock, *start* is the byte offset at which the lock starts, relative to *whence*, and *whence* is as with `io.IOBase.seek()`, specifically :
— `0` -- relative to the start of the file (`os.SEEK_SET`)
— `1` -- relative to the current buffer position (`os.SEEK_CUR`)
— `2` -- relative to the end of the file (`os.SEEK_END`)
The default for *start* is 0, which means to start at the beginning of the file. The default for *len* is 0 which means to lock to the end of the file. The default for *whence* is also 0.

Raises an *auditing event* `fcntl.lockf` with arguments `fd`, `cmd`, `len`, `start`, `whence`.

Examples (all on a SVR4 compliant system) :

```
import struct, fcntl, os

f = open(...)
rv = fcntl.fcntl(f, fcntl.F_SETFL, os.O_NDELAY)

lockdata = struct.pack('hhllhh', fcntl.F_WRLCK, 0, 0, 0, 0, 0)
rv = fcntl.fcntl(f, fcntl.F_SETLKW, lockdata)
```

Note that in the first example the return value variable *rv* will hold an integer value ; in the second example it will hold a `bytes` object. The structure lay-out for the *lockdata* variable is system dependent --- therefore using the `flock()` call may be better.

Voir aussi

Module `os`
If the locking flags `O_SHLOCK` and `O_EXLOCK` are present in the `os` module (on BSD only), the `os.open()` function provides an alternative to the `lockf()` and `flock()` functions.

35.8 `resource` --- Resource usage information

This module provides basic mechanisms for measuring and controlling system resources utilized by a program.

Availability : Unix, not WASI.

Symbolic constants are used to specify particular system resources and to request usage information about either the current process or its children.

An `OSError` is raised on syscall failure.

exception `resource.error`
A deprecated alias of `OSError`.

Modifié dans la version 3.3 : Following **PEP 3151**, this class was made an alias of `OSError`.

35.8.1 Resource Limits

Resources usage can be limited using the `setrlimit()` function described below. Each resource is controlled by a pair of limits : a soft limit and a hard limit. The soft limit is the current limit, and may be lowered or raised by a process over time. The soft limit can never exceed the hard limit. The hard limit can be lowered to any value greater than the soft limit, but not raised. (Only processes with the effective UID of the super-user can raise a hard limit.)

The specific resources that can be limited are system dependent. They are described in the `getrlimit(2)` man page. The resources listed below are supported when the underlying operating system supports them ; resources which cannot be checked or controlled by the operating system are not defined in this module for those platforms.

`resource.`**`RLIM_INFINITY`**

Constant used to represent the limit for an unlimited resource.

`resource.`**`getrlimit`**(*resource*)

Returns a tuple `(soft, hard)` with the current soft and hard limits of *resource*. Raises `ValueError` if an invalid resource is specified, or `error` if the underlying system call fails unexpectedly.

`resource.`**`setrlimit`**(*resource*, *limits*)

Sets new limits of consumption of *resource*. The *limits* argument must be a tuple `(soft, hard)` of two integers describing the new limits. A value of `RLIM_INFINITY` can be used to request a limit that is unlimited.

Raises `ValueError` if an invalid resource is specified, if the new soft limit exceeds the hard limit, or if a process tries to raise its hard limit. Specifying a limit of `RLIM_INFINITY` when the hard or system limit for that resource is not unlimited will result in a `ValueError`. A process with the effective UID of super-user can request any valid limit value, including unlimited, but `ValueError` will still be raised if the requested limit exceeds the system imposed limit.

`setrlimit` may also raise `error` if the underlying system call fails.

VxWorks only supports setting `RLIMIT_NOFILE`.

Raises an *auditing event* `resource.setrlimit` with arguments `resource`, `limits`.

`resource.`**`prlimit`**(*pid*, *resource*[, *limits*])

Combines `setrlimit()` and `getrlimit()` in one function and supports to get and set the resources limits of an arbitrary process. If *pid* is 0, then the call applies to the current process. *resource* and *limits* have the same meaning as in `setrlimit()`, except that *limits* is optional.

When *limits* is not given the function returns the *resource* limit of the process *pid*. When *limits* is given the *resource* limit of the process is set and the former resource limit is returned.

Raises `ProcessLookupError` when *pid* can't be found and `PermissionError` when the user doesn't have `CAP_SYS_RESOURCE` for the process.

Raises an *auditing event* `resource.prlimit` with arguments `pid`, `resource`, `limits`.

Availability : Linux >= 2.6.36 with glibc >= 2.13.

Ajouté dans la version 3.4.

These symbols define resources whose consumption can be controlled using the `setrlimit()` and `getrlimit()` functions described below. The values of these symbols are exactly the constants used by C programs.

The Unix man page for `getrlimit(2)` lists the available resources. Note that not all systems use the same symbol or same value to denote the same resource. This module does not attempt to mask platform differences --- symbols not defined for a platform will not be available from this module on that platform.

`resource.`**`RLIMIT_CORE`**

The maximum size (in bytes) of a core file that the current process can create. This may result in the creation of a partial core file if a larger core would be required to contain the entire process image.

`resource.`**`RLIMIT_CPU`**

The maximum amount of processor time (in seconds) that a process can use. If this limit is exceeded, a `SIGXCPU` signal is sent to the process. (See the `signal` module documentation for information about how to catch this signal and do something useful, e.g. flush open files to disk.)

`resource.RLIMIT_FSIZE`
The maximum size of a file which the process may create.

`resource.RLIMIT_DATA`
The maximum size (in bytes) of the process's heap.

`resource.RLIMIT_STACK`
The maximum size (in bytes) of the call stack for the current process. This only affects the stack of the main thread in a multi-threaded process.

`resource.RLIMIT_RSS`
The maximum resident set size that should be made available to the process.

`resource.RLIMIT_NPROC`
The maximum number of processes the current process may create.

`resource.RLIMIT_NOFILE`
The maximum number of open file descriptors for the current process.

`resource.RLIMIT_OFILE`
The BSD name for *RLIMIT_NOFILE*.

`resource.RLIMIT_MEMLOCK`
The maximum address space which may be locked in memory.

`resource.RLIMIT_VMEM`
The largest area of mapped memory which the process may occupy.

Availability : FreeBSD >= 11.

`resource.RLIMIT_AS`
The maximum area (in bytes) of address space which may be taken by the process.

`resource.RLIMIT_MSGQUEUE`
The number of bytes that can be allocated for POSIX message queues.

Availability : Linux >= 2.6.8.

Ajouté dans la version 3.4.

`resource.RLIMIT_NICE`
The ceiling for the process's nice level (calculated as 20 - rlim_cur).

Availability : Linux >= 2.6.12.

Ajouté dans la version 3.4.

`resource.RLIMIT_RTPRIO`
The ceiling of the real-time priority.

Availability : Linux >= 2.6.12.

Ajouté dans la version 3.4.

`resource.RLIMIT_RTTIME`
The time limit (in microseconds) on CPU time that a process can spend under real-time scheduling without making a blocking syscall.

Availability : Linux >= 2.6.25.

Ajouté dans la version 3.4.

`resource.RLIMIT_SIGPENDING`
The number of signals which the process may queue.

Availability : Linux >= 2.6.8.

Ajouté dans la version 3.4.

`resource.RLIMIT_SBSIZE`

The maximum size (in bytes) of socket buffer usage for this user. This limits the amount of network memory, and hence the amount of mbufs, that this user may hold at any time.

Availability : FreeBSD.

Ajouté dans la version 3.4.

`resource.RLIMIT_SWAP`

The maximum size (in bytes) of the swap space that may be reserved or used by all of this user id's processes. This limit is enforced only if bit 1 of the vm.overcommit sysctl is set. Please see tuning(7) for a complete description of this sysctl.

Availability : FreeBSD.

Ajouté dans la version 3.4.

`resource.RLIMIT_NPTS`

The maximum number of pseudo-terminals created by this user id.

Availability : FreeBSD.

Ajouté dans la version 3.4.

`resource.RLIMIT_KQUEUES`

The maximum number of kqueues this user id is allowed to create.

Availability : FreeBSD >= 11.

Ajouté dans la version 3.10.

35.8.2 Resource Usage

These functions are used to retrieve resource usage information :

`resource.getrusage(who)`

This function returns an object that describes the resources consumed by either the current process or its children, as specified by the *who* parameter. The *who* parameter should be specified using one of the `RUSAGE_*` constants described below.

Un exemple simple :

```
from resource import *
import time

# a non CPU-bound task
time.sleep(3)
print(getrusage(RUSAGE_SELF))

# a CPU-bound task
for i in range(10 ** 8):
    _ = 1 + 1
print(getrusage(RUSAGE_SELF))
```

The fields of the return value each describe how a particular system resource has been used, e.g. amount of time spent running is user mode or number of times the process was swapped out of main memory. Some values are dependent on the clock tick internal, e.g. the amount of memory the process is using.

For backward compatibility, the return value is also accessible as a tuple of 16 elements.

The fields `ru_utime` and `ru_stime` of the return value are floating-point values representing the amount of time spent executing in user mode and the amount of time spent executing in system mode, respectively. The remaining values are integers. Consult the *getrusage(2)* man page for detailed information about these values. A brief summary is presented here :

Index	Champ	Resource
0	ru_utime	time in user mode (float seconds)
1	ru_stime	time in system mode (float seconds)
2	ru_maxrss	maximum resident set size
3	ru_ixrss	shared memory size
4	ru_idrss	unshared memory size
5	ru_isrss	unshared stack size
6	ru_minflt	page faults not requiring I/O
7	ru_majflt	page faults requiring I/O
8	ru_nswap	number of swap outs
9	ru_inblock	block input operations
10	ru_oublock	block output operations
11	ru_msgsnd	messages sent
12	ru_msgrcv	messages received
13	ru_nsignals	signals received
14	ru_nvcsw	voluntary context switches
15	ru_nivcsw	involuntary context switches

This function will raise a `ValueError` if an invalid *who* parameter is specified. It may also raise `error` exception in unusual circumstances.

`resource.`**`getpagesize`**`()`

Returns the number of bytes in a system page. (This need not be the same as the hardware page size.)

The following `RUSAGE_*` symbols are passed to the `getrusage()` function to specify which processes information should be provided for.

`resource.`**`RUSAGE_SELF`**

Pass to `getrusage()` to request resources consumed by the calling process, which is the sum of resources used by all threads in the process.

`resource.`**`RUSAGE_CHILDREN`**

Pass to `getrusage()` to request resources consumed by child processes of the calling process which have been terminated and waited for.

`resource.`**`RUSAGE_BOTH`**

Pass to `getrusage()` to request resources consumed by both the current process and child processes. May not be available on all systems.

`resource.`**`RUSAGE_THREAD`**

Pass to `getrusage()` to request resources consumed by the current thread. May not be available on all systems.

Ajouté dans la version 3.2.

35.9 `syslog` --- Unix syslog library routines

Ce module fournit une interface aux routines de la bibliothèque `syslog` Unix. Consultez les pages du manuel Unix pour plus de détails sur `syslog`.

Availability : Unix, not WASI, not iOS.

This module wraps the system `syslog` family of routines. A pure Python library that can speak to a syslog server is available in the `logging.handlers` module as `SysLogHandler`.

Le module définit les fonctions suivantes :

`syslog.`**`syslog`**`(`*message*`)`

syslog.**syslog**(*priority*, *message*)

Envoie la chaîne de caractères *message* au journaliseur système. Un retour à la ligne est ajouté à la fin du message si nécessaire. Chaque message est marqué avec une priorité constituée d'une *fonction* et d'un *niveau*. L'argument optionnel *priority*, qui par défaut est `LOG_INFO`, définit la priorité du message. Si la fonction n'est pas encodée dans *priority* en utilisant un OU logique (`LOG_INFO | LOG_USER`), la valeur donnée lors de l'appel à `openlog()` est utilisée.

If `openlog()` has not been called prior to the call to `syslog()`, `openlog()` will be called with no arguments.

Lève un *évènement d'audit* `syslog.syslog` avec les arguments `priority`, `message`.

Modifié dans la version 3.2 : In previous versions, `openlog()` would not be called automatically if it wasn't called prior to the call to `syslog()`, deferring to the syslog implementation to call `openlog()`.

Modifié dans la version 3.12 : This function is restricted in subinterpreters. (Only code that runs in multiple interpreters is affected and the restriction is not relevant for most users.) `openlog()` must be called in the main interpreter before `syslog()` may be used in a subinterpreter. Otherwise it will raise `RuntimeError`.

syslog.**openlog**([*ident*[, *logoption*[, *facility*]]])

Les options de journalisation utilisées lors des appels à `syslog()` peuvent être définies en appelant `openlog()`. `syslog()` appellera `openlog()` sans argument si le journal n'est pas déjà ouvert.

L'argument nommé optionnel *ident* est une chaîne de caractères qui est ajoutée au début de chaque message. Par défaut, le dernier élément du chemin définit dans `sys.argv[0]` est utilisé. L'argument nommé optionnel *logoption* est un champ de bits (défini à 0 par défaut) -- Voir ci-dessous pour les combinaisons possibles. L'argument nommé optionnel *facility* définit la fonction à utiliser pour les messages qui n'en définissent pas. Par défaut, `LOG_USER` est utilisée.

Lève un *évènement d'audit* `syslog.openlog` avec les arguments `ident`, `logoption`, `facility`.

Modifié dans la version 3.2 : In previous versions, keyword arguments were not allowed, and *ident* was required.

Modifié dans la version 3.12 : This function is restricted in subinterpreters. (Only code that runs in multiple interpreters is affected and the restriction is not relevant for most users.) This may only be called in the main interpreter. It will raise `RuntimeError` if called in a subinterpreter.

syslog.**closelog**()

Réinitialise la configuration du module *syslog* et appelle la bibliothèque système `closelog()`.

Cet appel permet au module de se comporter comme lors de son import initial. Par exemple, `openlog()` sera appelée lors du premier appel à `syslog()` (sauf si `openlog()` a déjà été appelée). Quant à *ident* et aux paramètres de `openlog()`, ceux-ci seront réinitialisés à leur valeur par défaut.

Lève un *évènement d'audit* `syslog.closelog` sans argument.

Modifié dans la version 3.12 : This function is restricted in subinterpreters. (Only code that runs in multiple interpreters is affected and the restriction is not relevant for most users.) This may only be called in the main interpreter. It will raise `RuntimeError` if called in a subinterpreter.

syslog.**setlogmask**(*maskpri*)

Définit le masque de priorité à la valeur *maskpri* et retourne la précédente valeur du masque. Les appels à `syslog()` avec un niveau de priorité non présent dans *maskpri* seront ignorés. La fonction `LOG_MASK(pri)` calcule le masque pour la priorité *pri*. La fonction `LOG_UPTO(pri)` calcule le masque pour toutes les priorités jusqu'à *pri* (inclus).

Lève un *évènement d'audit* `syslog.setlogmask` avec l'argument `maskpri`.

Le module définit les constantes suivantes :

syslog.**LOG_EMERG**

syslog.**LOG_ALERT**

syslog.**LOG_CRIT**

syslog.**LOG_ERR**

syslog.**LOG_WARNING**

syslog.**LOG_NOTICE**

syslog.**LOG_INFO**

syslog.**LOG_DEBUG**

Priority levels (high to low).

syslog.**LOG_AUTH**

syslog.**LOG_AUTHPRIV**

syslog.**LOG_CRON**

syslog.**LOG_DAEMON**

syslog.**LOG_FTP**

syslog.**LOG_INSTALL**

syslog.**LOG_KERN**

syslog.**LOG_LAUNCHD**

syslog.**LOG_LPR**

syslog.**LOG_MAIL**

syslog.**LOG_NETINFO**

syslog.**LOG_NEWS**

syslog.**LOG_RAS**

syslog.**LOG_REMOTEAUTH**

syslog.**LOG_SYSLOG**

syslog.**LOG_USER**

syslog.**LOG_UUCP**

syslog.**LOG_LOCAL0**

syslog.**LOG_LOCAL1**

syslog.**LOG_LOCAL2**

syslog.**LOG_LOCAL3**

syslog.**LOG_LOCAL4**

syslog.**LOG_LOCAL5**

syslog.**LOG_LOCAL6**

syslog.**LOG_LOCAL7**

Facilities, depending on availability in `<syslog.h>` for *LOG_AUTHPRIV*, *LOG_FTP*, *LOG_NETINFO*, *LOG_REMOTEAUTH*, *LOG_INSTALL* and *LOG_RAS*.

Modifié dans la version 3.13 : Added *LOG_FTP*, *LOG_NETINFO*, *LOG_REMOTEAUTH*, *LOG_INSTALL*, *LOG_RAS*, and *LOG_LAUNCHD*.

syslog.**LOG_PID**

syslog.**LOG_CONS**

syslog.**LOG_NDELAY**

syslog.**LOG_ODELAY**

syslog.**LOG_NOWAIT**

syslog.**LOG_PERROR**

Log options, depending on availability in `<syslog.h>` for *LOG_ODELAY*, *LOG_NOWAIT* and *LOG_PERROR*.

35.9.1 Exemples

Exemple simple

Un simple jeu d'exemples :

```
import syslog

syslog.syslog('Processing started')
if error:
    syslog.syslog(syslog.LOG_ERR, 'Processing started')
```

Un exemple montrant comment définir certaines options de journalisation. Ces options ajoutent l'identifiant du processus (*PID*) dans les messages journalisés et écrivent ces messages à l'aide de la fonction utilisée pour la journalisation des systèmes de messagerie :

```
syslog.openlog(logoption=syslog.LOG_PID, facility=syslog.LOG_MAIL)
syslog.syslog('E-mail processing initiated...')
```

CHAPITRE 36

Modules command-line interface (CLI)

The following modules have a command-line interface.

- *ast*
- *asyncio*
- `base64`
- *calendar*
- `code`
- *compileall*
- `cProfile` : see *profile*
- *difflib*
- *dis*
- `doctest`
- `encodings.rot_13`
- `ensurepip`
- `filecmp`
- `fileinput`
- `ftplib`
- *gzip*
- *http.server*
- `idlelib`
- *inspect*
- *json*
- `mimetypes`
- `pdb`
- `pickle`
- *pickletools*
- `platform`
- `poplib`
- *profile*
- `pstats`
- *py_compile*
- `pyclbr`
- `pydoc`
- `quopri`
- *random*
- `runpy`
- *site*

- *sqlite3*
- *symtable*
- *sysconfig*
- `tabnanny`
- *tarfile*
- `this`
- *timeit*
- *tokenize*
- *trace*
- `turtledemo`
- *unittest*
- *uuid*
- `venv`
- `webbrowser`
- *zipapp*
- *zipfile*

See also the Python command-line interface.

CHAPITRE 37

Modules remplacés

The modules described in this chapter have been superseded by other modules for most use cases, and are retained primarily to preserve backwards compatibility.

Modules may appear in this chapter because they only cover a limited subset of a problem space, and a more generally applicable solution is available elsewhere in the standard library (for example, `getopt` covers the very specific task of "mimic the C `getopt()` API in Python", rather than the broader command line option parsing and argument parsing capabilities offered by `optparse` and `argparse`).

Alternatively, modules may appear in this chapter because they are deprecated outright, and awaiting removal in a future release, or they are *soft deprecated* and their use is actively discouraged in new projects. With the removal of various obsolete modules through **PEP 594**, there are currently no modules in this latter category.

37.1 `getopt` --- C-style parser for command line options

Code source : Lib/getopt.py

> **Note**
>
> This module is considered feature complete. A more declarative and extensible alternative to this API is provided in the `optparse` module. Further functional enhancements for command line parameter processing are provided either as third party modules on PyPI, or else as features in the `argparse` module.

This module helps scripts to parse the command line arguments in `sys.argv`. It supports the same conventions as the Unix `getopt()` function (including the special meanings of arguments of the form '-' and '--'). Long options similar to those supported by GNU software may be used as well via an optional third argument.

Users who are unfamiliar with the Unix `getopt()` function should consider using the `argparse` module instead. Users who are familiar with the Unix `getopt()` function, but would like to get equivalent behavior while writing less code and getting better help and error messages should consider using the `optparse` module. See *Choosing an argument parsing library* for additional details.

Ce module fournit deux fonctions et une exception :

getopt.**getopt**(*args*, *shortopts*, *longopts=[]*)

Parses command line options and parameter list. *args* is the argument list to be parsed, without the leading reference to the running program. Typically, this means `sys.argv[1:]`. *shortopts* is the string of option letters that the script wants to recognize, with options that require an argument followed by a colon (`':'`) and options that accept an optional argument followed by two colons (`'::'`); i.e., the same format that Unix `getopt()` uses.

> **Note**
>
> Unlike GNU `getopt()`, after a non-option argument, all further arguments are considered also non-options. This is similar to the way non-GNU Unix systems work.

longopts, if specified, must be a list of strings with the names of the long options which should be supported. The leading `'--'` characters should not be included in the option name. Long options which require an argument should be followed by an equal sign (`'='`). Long options which accept an optional argument should be followed by an equal sign and question mark (`'=?'`). To accept only long options, *shortopts* should be an empty string. Long options on the command line can be recognized so long as they provide a prefix of the option name that matches exactly one of the accepted options. For example, if *longopts* is `['foo', 'frob']`, the option `--fo` will match as `--foo`, but `--f` will not match uniquely, so `GetoptError` will be raised.

La valeur de retour se compose de deux éléments : le premier est une liste de paires `(option, value)`, la deuxième est la liste des arguments de programme laissés après que la liste d'options est été dépouillée (il s'agit d'une tranche de fin de *args*). Chaque paire option-valeur retournée a l'option comme premier élément, préfixée avec un trait d'union pour les options courtes (par exemple, `'-x'`) ou deux tirets pour les options longues (par exemple, `'--long-option'`), et l'argument option comme deuxième élément, ou une chaîne vide si le option n'a aucun argument. Les options se trouvent dans la liste dans l'ordre dans lequel elles ont été trouvées, permettant ainsi plusieurs occurrences. Les options longues et courtes peuvent être mélangées.

Modifié dans la version 3.14 : Optional arguments are supported.

getopt.**gnu_getopt**(*args*, *shortopts*, *longopts=[]*)

Cette fonction fonctionne comme `getopt()`, sauf que le mode de *scan* GNU est utilisé par défaut. Cela signifie que les arguments option et non-option peuvent être **intermixés**. La fonction `getopt()` arrête le traitement des options dès qu'un argument de non-option est rencontré.

If the first character of the option string is `'+'`, or if the environment variable `POSIXLY_CORRECT` is set, then option processing stops as soon as a non-option argument is encountered.

If the first character of the option string is `'-'`, non-option arguments that are followed by options are added to the list of option-and-value pairs as a pair that has `None` as its first element and the list of non-option arguments as its second element. The second element of the `gnu_getopt()` result is a list of program arguments after the last option.

Modifié dans la version 3.14 : Support for returning intermixed options and non-option arguments in order.

exception getopt.**GetoptError**

This is raised when an unrecognized option is found in the argument list or when an option requiring an argument is given none. The argument to the exception is a string indicating the cause of the error. For long options, an argument given to an option which does not require one will also cause this exception to be raised. The attributes `msg` and `opt` give the error message and related option; if there is no specific option to which the exception relates, `opt` is an empty string.

exception getopt.**error**

Alias pour `GetoptError`; pour la rétrocompatibilité.

Un exemple utilisant uniquement les options de style UNIX :

```
>>> import getopt
>>> args = '-a -b -cfoo -d bar a1 a2'.split()
>>> args
```

(suite sur la page suivante)

(suite de la page précédente)

```
['-a', '-b', '-cfoo', '-d', 'bar', 'a1', 'a2']
>>> optlist, args = getopt.getopt(args, 'abc:d:')
>>> optlist
[('-a', ''), ('-b', ''), ('-c', 'foo'), ('-d', 'bar')]
>>> args
['a1', 'a2']
```

L'utilisation de noms d'options longs est tout aussi simple :

```
>>> s = '--condition=foo --testing --output-file abc.def -x a1 a2'
>>> args = s.split()
>>> args
['--condition=foo', '--testing', '--output-file', 'abc.def', '-x', 'a1', 'a2']
>>> optlist, args = getopt.getopt(args, 'x', [
...     'condition=', 'output-file=', 'testing'])
>>> optlist
[('--condition', 'foo'), ('--testing', ''), ('--output-file', 'abc.def'), ('-x', '
↪')]
>>> args
['a1', 'a2']
```

Optional arguments should be specified explicitly :

```
>>> s = '-Con -C --color=off --color a1 a2'
>>> args = s.split()
>>> args
['-Con', '-C', '--color=off', '--color', 'a1', 'a2']
>>> optlist, args = getopt.getopt(args, 'C::', ['color=?'])
>>> optlist
[('-C', 'on'), ('-C', ''), ('--color', 'off'), ('--color', '')]
>>> args
['a1', 'a2']
```

The order of options and non-option arguments can be preserved :

```
>>> s = 'a1 -x a2 a3 a4 --long a5 a6'
>>> args = s.split()
>>> args
['a1', '-x', 'a2', 'a3', 'a4', '--long', 'a5', 'a6']
>>> optlist, args = getopt.gnu_getopt(args, '-x:', ['long='])
>>> optlist
[(None, ['a1']), ('-x', 'a2'), (None, ['a3', 'a4']), ('--long', 'a5')]
>>> args
['a6']
```

In a script, typical usage is something like this :

```
import getopt, sys

def main():
    try:
        opts, args = getopt.getopt(sys.argv[1:], "ho:v", ["help", "output="])
    except getopt.GetoptError as err:
        # print help information and exit:
        print(err)  # will print something like "option -a not recognized"
        usage()
        sys.exit(2)
```

(suite sur la page suivante)

(suite de la page précédente)

```
    output = None
    verbose = False
    for o, a in opts:
        if o == "-v":
            verbose = True
        elif o in ("-h", "--help"):
            usage()
            sys.exit()
        elif o in ("-o", "--output"):
            output = a
        else:
            assert False, "unhandled option"
    process(args, output=output, verbose=verbose)

if __name__ == "__main__":
    main()
```

Note that an equivalent command line interface could be produced with less code and more informative help and error messages by using the *optparse* module :

```
import optparse

if __name__ == '__main__':
    parser = optparse.OptionParser()
    parser.add_option('-o', '--output')
    parser.add_option('-v', dest='verbose', action='store_true')
    opts, args = parser.parse_args()
    process(args, output=opts.output, verbose=opts.verbose)
```

A roughly equivalent command line interface for this case can also be produced by using the *argparse* module :

```
import argparse

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('-o', '--output')
    parser.add_argument('-v', dest='verbose', action='store_true')
    parser.add_argument('rest', nargs='*')
    args = parser.parse_args()
    process(args.rest, output=args.output, verbose=args.verbose)
```

See *Choosing an argument parsing library* for details on how the `argparse` version of this code differs in behaviour from the `optparse` (and `getopt`) version.

Voir aussi

Module *optparse*
Declarative command line option parsing.

Module *argparse*
More opinionated command line option and argument parsing library.

CHAPITRE 38

Removed Modules

The modules described in this chapter have been removed from the Python standard library. They are documented here to help people find replacements.

38.1 `aifc` --- Read and write AIFF and AIFC files

Deprecated since version 3.11, removed in version 3.13.

This module is no longer part of the Python standard library. It was removed in Python 3.13 after being deprecated in Python 3.11. The removal was decided in **PEP 594**.

The last version of Python that provided the `aifc` module was Python 3.12.

38.2 `asynchat` --- Asynchronous socket command/response handler

Deprecated since version 3.6, removed in version 3.12.

This module is no longer part of the Python standard library. It was removed in Python 3.12 after being deprecated in Python 3.6. The removal was decided in **PEP 594**.

Applications should use the *`asyncio`* module instead.

The last version of Python that provided the `asynchat` module was Python 3.11.

38.3 `asyncore` --- Asynchronous socket handler

Deprecated since version 3.6, removed in version 3.12.

This module is no longer part of the Python standard library. It was removed in Python 3.12 after being deprecated in Python 3.6. The removal was decided in **PEP 594**.

Applications should use the *`asyncio`* module instead.

The last version of Python that provided the `asyncore` module was Python 3.11.

38.4 `audioop` --- Manipulate raw audio data

Deprecated since version 3.11, removed in version 3.13.

This module is no longer part of the Python standard library. It was removed in Python 3.13 after being deprecated in Python 3.11. The removal was decided in **PEP 594**.

The last version of Python that provided the `audioop` module was Python 3.12.

38.5 `cgi` --- Common Gateway Interface support

Deprecated since version 3.11, removed in version 3.13.

This module is no longer part of the Python standard library. It was removed in Python 3.13 after being deprecated in Python 3.11. The removal was decided in **PEP 594**.

A fork of the module on PyPI can be used instead : legacy-cgi. This is a copy of the cgi module, no longer maintained or supported by the core Python team.

The last version of Python that provided the `cgi` module was Python 3.12.

38.6 `cgitb` --- Traceback manager for CGI scripts

Deprecated since version 3.11, removed in version 3.13.

This module is no longer part of the Python standard library. It was removed in Python 3.13 after being deprecated in Python 3.11. The removal was decided in **PEP 594**.

A fork of the module on PyPI can now be used instead : legacy-cgi. This is a copy of the cgi module, no longer maintained or supported by the core Python team.

The last version of Python that provided the `cgitb` module was Python 3.12.

38.7 `chunk` --- Read IFF chunked data

Deprecated since version 3.11, removed in version 3.13.

This module is no longer part of the Python standard library. It was removed in Python 3.13 after being deprecated in Python 3.11. The removal was decided in **PEP 594**.

The last version of Python that provided the `chunk` module was Python 3.12.

38.8 `crypt` --- Function to check Unix passwords

Deprecated since version 3.11, removed in version 3.13.

This module is no longer part of the Python standard library. It was removed in Python 3.13 after being deprecated in Python 3.11. The removal was decided in **PEP 594**.

Applications can use the *`hashlib`* module from the standard library. Other possible replacements are third-party libraries from PyPI : legacycrypt, bcrypt, argon2-cffi, or passlib. These are not supported or maintained by the Python core team.

The last version of Python that provided the `crypt` module was Python 3.12.

38.9 `distutils` --- Building and installing Python modules

Deprecated since version 3.10, removed in version 3.12.

This module is no longer part of the Python standard library. It was removed in Python 3.12 after being deprecated in Python 3.10. The removal was decided in **PEP 632**, which has migration advice.

The last version of Python that provided the `distutils` module was Python 3.11.

38.10 `imghdr` --- Determine the type of an image

Deprecated since version 3.11, removed in version 3.13.

This module is no longer part of the Python standard library. It was removed in Python 3.13 after being deprecated in Python 3.11. The removal was decided in **PEP 594**.

Possible replacements are third-party libraries from PyPI : filetype, puremagic, or python-magic. These are not supported or maintained by the Python core team.

The last version of Python that provided the `imghdr` module was Python 3.12.

38.11 `imp` --- Access the import internals

Deprecated since version 3.4, removed in version 3.12.

This module is no longer part of the Python standard library. It was removed in Python 3.12 after being deprecated in Python 3.4.

The removal notice includes guidance for migrating code from `imp` to *`importlib`*.

The last version of Python that provided the `imp` module was Python 3.11.

38.12 `mailcap` --- Mailcap file handling

Deprecated since version 3.11, removed in version 3.13.

This module is no longer part of the Python standard library. It was removed in Python 3.13 after being deprecated in Python 3.11. The removal was decided in **PEP 594**.

The last version of Python that provided the `mailcap` module was Python 3.12.

38.13 `msilib` --- Read and write Microsoft Installer files

Deprecated since version 3.11, removed in version 3.13.

This module is no longer part of the Python standard library. It was removed in Python 3.13 after being deprecated in Python 3.11. The removal was decided in **PEP 594**.

The last version of Python that provided the `msilib` module was Python 3.12.

38.14 `nis` --- Interface to Sun's NIS (Yellow Pages)

Deprecated since version 3.11, removed in version 3.13.

This module is no longer part of the Python standard library. It was removed in Python 3.13 after being deprecated in Python 3.11. The removal was decided in **PEP 594**.

The last version of Python that provided the `nis` module was Python 3.12.

38.15 `nntplib` --- NNTP protocol client

Deprecated since version 3.11, removed in version 3.13.

This module is no longer part of the Python standard library. It was removed in Python 3.13 after being deprecated in Python 3.11. The removal was decided in **PEP 594**.

The last version of Python that provided the `nntplib` module was Python 3.12.

38.16 ossaudiodev --- Access to OSS-compatible audio devices

Deprecated since version 3.11, removed in version 3.13.

This module is no longer part of the Python standard library. It was removed in Python 3.13 after being deprecated in Python 3.11. The removal was decided in **PEP 594**.

The last version of Python that provided the `ossaudiodev` module was Python 3.12.

38.17 pipes --- Interface to shell pipelines

Deprecated since version 3.11, removed in version 3.13.

This module is no longer part of the Python standard library. It was removed in Python 3.13 after being deprecated in Python 3.11. The removal was decided in **PEP 594**.

Applications should use the *subprocess* module instead.

The last version of Python that provided the `pipes` module was Python 3.12.

38.18 smtpd --- SMTP Server

Deprecated since version 3.6, removed in version 3.12.

This module is no longer part of the Python standard library. It was removed in Python 3.12 after being deprecated in Python 3.6. The removal was decided in **PEP 594**.

A possible replacement is the third-party aiosmtpd library. This library is not maintained or supported by the Python core team.

The last version of Python that provided the `smtpd` module was Python 3.11.

38.19 sndhdr --- Determine type of sound file

Deprecated since version 3.11, removed in version 3.13.

This module is no longer part of the Python standard library. It was removed in Python 3.13 after being deprecated in Python 3.11. The removal was decided in **PEP 594**.

Possible replacements are third-party modules from PyPI : filetype, puremagic, or python-magic. These are not supported or maintained by the Python core team.

The last version of Python that provided the `sndhdr` module was Python 3.12.

38.20 spwd --- The shadow password database

Deprecated since version 3.11, removed in version 3.13.

This module is no longer part of the Python standard library. It was removed in Python 3.13 after being deprecated in Python 3.11. The removal was decided in **PEP 594**.

A possible replacement is the third-party library python-pam. This library is not supported or maintained by the Python core team.

The last version of Python that provided the `spwd` module was Python 3.12.

38.21 sunau --- Read and write Sun AU files

Deprecated since version 3.11, removed in version 3.13.

This module is no longer part of the Python standard library. It was removed in Python 3.13 after being deprecated in Python 3.11. The removal was decided in **PEP 594**.

The last version of Python that provided the `sunau` module was Python 3.12.

38.22 `telnetlib` --- Telnet client

Deprecated since version 3.11, removed in version 3.13.

This module is no longer part of the Python standard library. It was removed in Python 3.13 after being deprecated in Python 3.11. The removal was decided in **PEP 594**.

Possible replacements are third-party libraries from PyPI : telnetlib3 or Exscript. These are not supported or maintained by the Python core team.

The last version of Python that provided the `telnetlib` module was Python 3.12.

38.23 `uu` --- Encode and decode uuencode files

Deprecated since version 3.11, removed in version 3.13.

This module is no longer part of the Python standard library. It was removed in Python 3.13 after being deprecated in Python 3.11. The removal was decided in **PEP 594**.

The last version of Python that provided the `uu` module was Python 3.12.

38.24 `xdrlib` --- Encode and decode XDR data

Deprecated since version 3.11, removed in version 3.13.

This module is no longer part of the Python standard library. It was removed in Python 3.13 after being deprecated in Python 3.11. The removal was decided in **PEP 594**.

The last version of Python that provided the `xdrlib` module was Python 3.12.

CHAPITRE 39

Considérations de sécurité

The following modules have specific security considerations :

- `base64` : *base64 security considerations* in **RFC 4648**
- `hashlib` : *all constructors take a "usedforsecurity" keyword-only argument disabling known insecure and blocked algorithms*
- `http.server` is not suitable for production use, only implementing basic security checks. See the *security considerations*.
- `logging` : *Logging configuration uses eval()*
- `multiprocessing` : *Connection.recv() uses pickle*
- `pickle` : *Restricting globals in pickle*
- `random` shouldn't be used for security purposes, use `secrets` instead
- `shelve` : *shelve is based on pickle and thus unsuitable for dealing with untrusted sources*
- `ssl` : *SSL/TLS security considerations*
- `subprocess` : *Subprocess security considerations*
- `tempfile` : *mktemp is deprecated due to vulnerability to race conditions*
- `xml` : *XML vulnerabilities*
- `zipfile` : *maliciously prepared .zip files can cause disk volume exhaustion*

The `-I` command line option can be used to run Python in isolated mode. When it cannot be used, the `-P` option or the `PYTHONSAFEPATH` environment variable can be used to not prepend a potentially unsafe path to `sys.path` such as the current directory, the script's directory or an empty string.

ANNEXE A

Glossaire

>>>

The default Python prompt of the *interactive* shell. Often seen for code examples which can be executed interactively in the interpreter.

...

Peut faire référence à :

— The default Python prompt of the *interactive* shell when entering the code for an indented code block, when within a pair of matching left and right delimiters (parentheses, square brackets, curly braces or triple quotes), or after specifying a decorator.

— La constante `Ellipsis`.

classe mère abstraite

Les classes mères abstraites (ABC, suivant l'abréviation anglaise *Abstract Base Class*) complètent le *duck-typing* en fournissant un moyen de définir des interfaces pour les cas où d'autres techniques comme `hasattr()` seraient inélégantes ou subtilement fausses (par exemple avec les méthodes magiques). Les ABC introduisent des sous-classes virtuelles qui n'héritent pas d'une classe mais qui sont quand même reconnues par `isinstance()` ou `issubclass()` (voir la documentation du module `abc`). Python contient de nombreuses ABC pour les structures de données (dans le module `collections.abc`), les nombres (dans le module `numbers`), les flux (dans le module `io`) et les chercheurs-chargeurs du système d'importation (dans le module `importlib.abc`). Vous pouvez créer vos propres ABC avec le module `abc`.

annotate function

A function that can be called to retrieve the *annotations* of an object. This function is accessible as the `__annotate__` attribute of functions, classes, and modules. Annotate functions are a subset of *evaluate functions*.

annotation

Étiquette associée à une variable, un attribut de classe, un paramètre de fonction ou une valeur de retour. Elle est utilisée par convention comme *type hint*.

Annotations of local variables cannot be accessed at runtime, but annotations of global variables, class attributes, and functions can be retrieved by calling `annotationlib.get_annotations()` on modules, classes, and functions, respectively.

See *variable annotation*, *function annotation*, **PEP 484**, **PEP 526**, and **PEP 649**, which describe this functionality. Also see annotations-howto for best practices on working with annotations.

argument

Valeur, donnée à une *fonction* ou à une *méthode* lors de son appel. Il existe deux types d'arguments :

— *argument nommé* : un argument précédé d'un identifiant (comme `name=`) ou un dictionnaire précédé de `**`, lors d'un appel de fonction. Par exemple, `3` et `5` sont tous les deux des arguments nommés dans l'appel à `complex()` ici :

```
complex(real=3, imag=5)
complex(**{'real': 3, 'imag': 5})
```

— *argument positionnel* : un argument qui n'est pas nommé. Les arguments positionnels apparaissent au début de la liste des arguments, ou donnés sous forme d'un *itérable* précédé par `*`. Par exemple, `3` et `5` sont tous les deux des arguments positionnels dans les appels suivants :

```
complex(3, 5)
complex(*(3, 5))
```

Les arguments se retrouvent dans le corps de la fonction appelée parmi les variables locales. Voir la section calls à propos des règles dictant cette affectation. Syntaxiquement, toute expression est acceptée comme argument, et c'est la valeur résultante de l'expression qui sera affectée à la variable locale.

Voir aussi *paramètre* dans le glossaire, la question Différence entre argument et paramètre de la FAQ et la **PEP 362**.

gestionnaire de contexte asynchrone

(*asynchronous context manager* en anglais) Objet contrôlant l'environnement à l'intérieur d'une instruction `async with` en définissant les méthodes `__aenter__()` et `__aexit__()`. A été Introduit par la **PEP 492**.

générateur asynchrone

Fonction qui renvoie un *itérateur de générateur asynchrone*. Cela ressemble à une coroutine définie par `async def`, sauf qu'elle contient une ou des expressions `yield` produisant ainsi uns série de valeurs utilisables dans une boucle `async for`.

Générateur asynchrone fait généralement référence à une fonction, mais peut faire référence à un *itérateur de générateur asynchrone* dans certains contextes. Dans les cas où le sens voulu n'est pas clair, utiliser l'ensemble des termes lève l'ambiguïté.

Un générateur asynchrone peut contenir des expressions `await` ainsi que des instructions `async for`, et `async with`.

itérateur de générateur asynchrone

Objet créé par un *générateur asynchrone*.

C'est un *asynchronous iterator* qui, lorsqu'il est appelé via la méthode `__anext__()` renvoie un objet *awaitable* qui exécute le corps de la fonction du générateur asynchrone jusqu'au prochain `yield`.

Each `yield` temporarily suspends processing, remembering the execution state (including local variables and pending try-statements). When the *asynchronous generator iterator* effectively resumes with another awaitable returned by `__anext__()`, it picks up where it left off. See **PEP 492** and **PEP 525**.

itérable asynchrone

Objet qui peut être utilisé dans une instruction `async for`. Sa méthode `__aiter__()` doit renvoyer un *asynchronous iterator*. A été introduit par la **PEP 492**.

itérateur asynchrone

Objet qui implémente les méthodes `__aiter__()` et `__anext__()`. `__anext__()` doit renvoyer un objet *awaitable*. Tant que la méthode `__anext__()` produit des objets *awaitable*, le `async for` appelant les consomme. L'itérateur asynchrone lève une exception `StopAsyncIteration` pour signifier la fin de l'itération. A été introduit par la **PEP 492**.

attribut

Valeur associée à un objet et habituellement désignée par son nom *via* une notation utilisant des points. Par exemple, si un objet *o* possède un attribut *a*, cet attribut est référencé par *o.a*.

Il est possible de donner à un objet un attribut dont le nom n'est pas un identifiant tel que défini pour les identifiers, par exemple en utilisant `setattr()`, si l'objet le permet. Un tel attribut ne sera pas accessible à l'aide d'une expression pointée et on devra y accéder avec `getattr()`.

attendable (*awaitable*)

Objet pouvant être utilisé dans une expression `await`. Ce peut être une *coroutine* ou un objet avec une méthode `__await__()`. Voir aussi la **PEP 492**.

BDFL

Dictateur bienveillant à vie (*Benevolent Dictator For Life* en anglais). Pseudonyme de Guido van Rossum, le créateur de Python.

fichier binaire

A *file object* able to read and write *bytes-like objects*. Examples of binary files are files opened in binary mode (`'rb'`, `'wb'` or `'rb+'`), `sys.stdin.buffer`, `sys.stdout.buffer`, and instances of `io.BytesIO` and `gzip.GzipFile`.

Consultez *fichier texte*, un objet fichier capable de lire et d'écrire des objets `str`.

référence empruntée

In Python's C API, a borrowed reference is a reference to an object, where the code using the object does not own the reference. It becomes a dangling pointer if the object is destroyed. For example, a garbage collection can remove the last *strong reference* to the object and so destroy it.

Il est recommandé d'appeler `Py_INCREF()` sur la *référence empruntée*, ce qui la transforme *in situ* en une *référence forte*. Vous pouvez faire une exception si vous êtes certain que l'objet ne peut pas être supprimé avant la dernière utilisation de la référence empruntée. Voir aussi la fonction `Py_NewRef()`, qui crée une nouvelle *référence forte*.

objet octet-compatible

Un objet gérant le protocole tampon et pouvant exporter un tampon (*buffer* en anglais) C-*contigu*. Cela inclut les objets `bytes`, `bytearray` et `array.array`, ainsi que beaucoup d'objets `memoryview`. Les objets octets-compatibles peuvent être utilisés pour diverses opérations sur des données binaires, comme la compression, la sauvegarde dans un fichier binaire ou l'envoi sur le réseau.

Certaines opérations nécessitent de travailler sur des données binaires variables. La documentation parle de ceux-ci comme des *read-write bytes-like objects*. Par exemple, `bytearray` ou une `memoryview` d'un `bytearray` en font partie. D'autres opérations nécessitent de travailler sur des données binaires stockées dans des objets immuables (« *objets octets-compatibles en lecture seule* »), par exemple des `bytes` ou des `memoryview` d'un objet `bytes`.

code intermédiaire (*bytecode*)

Le code source, en Python, est compilé en un code intermédiaire (*bytecode* en anglais), la représentation interne à CPython d'un programme Python. Le code intermédiaire est mis en cache dans un fichier `.pyc` de manière à ce qu'une seconde exécution soit plus rapide (la compilation en code intermédiaire a déjà été faite). On dit que ce *langage intermédiaire* est exécuté sur une *virtual machine* qui exécute des instructions machine pour chaque instruction du code intermédiaire. Notez que le code intermédiaire n'a pas vocation à fonctionner sur différentes machines virtuelles Python ou à être stable entre différentes versions de Python.

La documentation du *module dis* fournit une liste des instructions du code intermédiaire.

appelable (*callable*)

Un appelable est un objet qui peut être appelé, éventuellement avec un ensemble d'arguments (voir *argument*), avec la syntaxe suivante :

```
callable(argument1, argument2, argumentN)
```

Une *fonction*, et par extension une *méthode*, est un appelable. Une instance d'une classe qui implémente la méthode `__call__()` est également un appelable.

fonction de rappel (*callback*)

Une fonction (classique, par opposition à une coroutine) passée en argument pour être exécutée plus tard.

classe

Modèle pour créer des objets définis par l'utilisateur. Une définition de classe (*class*) contient normalement des définitions de méthodes qui agissent sur les instances de la classe.

variable de classe

Une variable définie dans une classe et destinée à être modifiée uniquement au niveau de la classe (c'est-à-dire, pas dans une instance de la classe).

closure variable

A *free variable* referenced from a *nested scope* that is defined in an outer scope rather than being resolved at runtime from the globals or builtin namespaces. May be explicitly defined with the `nonlocal` keyword to allow write access, or implicitly defined if the variable is only being read.

For example, in the `inner` function in the following code, both `x` and `print` are *free variables*, but only `x` is a *closure variable* :

```
def outer():
    x = 0
    def inner():
        nonlocal x
        x += 1
        print(x)
    return inner
```

Due to the `codeobject.co_freevars` attribute (which, despite its name, only includes the names of closure variables rather than listing all referenced free variables), the more general *free variable* term is sometimes used even when the intended meaning is to refer specifically to closure variables.

nombre complexe

Extension des nombres réels familiers, dans laquelle tous les nombres sont exprimés sous la forme d'une somme d'une partie réelle et d'une partie imaginaire. Les nombres imaginaires sont les nombres réels multipliés par l'unité imaginaire (la racine carrée de `-1`, souvent écrite `i` en mathématiques ou `j` par les ingénieurs). Python comprend nativement les nombres complexes, écrits avec cette dernière notation : la partie imaginaire est écrite avec un suffixe `j`, exemple, `3+1j`. Pour utiliser les équivalents complexes de `math`, utilisez `cmath`. Les nombres complexes sont un concept assez avancé en mathématiques. Si vous ne connaissez pas ce concept, vous pouvez tranquillement les ignorer.

context

This term has different meanings depending on where and how it is used. Some common meanings :

— The temporary state or environment established by a *context manager* via a `with` statement.

— The collection of keyvalue bindings associated with a particular `contextvars.Context` object and accessed via `ContextVar` objects. Also see *context variable*.

— A `contextvars.Context` object. Also see *current context*.

context management protocol

The `__enter__()` and `__exit__()` methods called by the `with` statement. See **PEP 343**.

gestionnaire de contexte

An object which implements the *context management protocol* and controls the environment seen in a `with` statement. See **PEP 343**.

variable de contexte

A variable whose value depends on which context is the *current context*. Values are accessed via `contextvars.ContextVar` objects. Context variables are primarily used to isolate state between concurrent asynchronous tasks.

contigu

Un tampon (*buffer* en anglais) est considéré comme contigu s'il est soit *C-contigu* soit *Fortran-contigu*. Les tampons de dimension zéro sont C-contigus et Fortran-contigus. Pour un tableau à une dimension, ses éléments doivent être placés en mémoire l'un à côté de l'autre, dans l'ordre croissant de leur indice, en commençant à zéro. Pour qu'un tableau multidimensionnel soit C-contigu, le dernier indice doit être celui qui varie le plus rapidement lors du parcours de ses éléments dans l'ordre de leur adresse mémoire. À l'inverse, dans les tableaux Fortran-contigu, c'est le premier indice qui doit varier le plus rapidement.

coroutine

Les coroutines sont une forme généralisée des fonctions. On entre dans une fonction en un point et on en sort en un autre point. On peut entrer, sortir et reprendre l'exécution d'une coroutine en plusieurs points. Elles peuvent être implémentées en utilisant l'instruction `async def`. Voir aussi la **PEP 492**.

fonction coroutine

Fonction qui renvoie un objet *coroutine*. Une fonction coroutine peut être définie par l'instruction `async def` et peut contenir les mots clés `await`, `async for` ainsi que `async with`. A été introduit par la **PEP 492**.

CPython

L'implémentation canonique du langage de programmation Python, tel que distribué sur python.org. Le terme "CPython" est utilisé dans certains contextes lorsqu'il est nécessaire de distinguer cette implémentation des autres comme *Jython* ou *IronPython*.

current context

The *context* (`contextvars.Context` object) that is currently used by `ContextVar` objects to access (get or set) the values of *context variables*. Each thread has its own current context. Frameworks for executing asynchronous tasks (see `asyncio`) associate each task with a context which becomes the current context whenever the task starts or resumes execution.

décorateur

Fonction dont la valeur de retour est une autre fonction. Un décorateur est habituellement utilisé pour transformer une fonction via la syntaxe `@wrapper`, dont les exemples typiques sont : `classmethod()` et `staticmethod()`.

La syntaxe des décorateurs est simplement du sucre syntaxique, les définitions des deux fonctions suivantes sont sémantiquement équivalentes :

```
def f(arg):
    ...
f = staticmethod(f)

@staticmethod
def f(arg):
    ...
```

Quoique moins fréquemment utilisé, le même concept existe pour les classes. Consultez la documentation définitions de fonctions et définitions de classes pour en savoir plus sur les décorateurs.

descripteur

Any object which defines the methods `__get__()`, `__set__()`, or `__delete__()`. When a class attribute is a descriptor, its special binding behavior is triggered upon attribute lookup. Normally, using *a.b* to get, set or delete an attribute looks up the object named *b* in the class dictionary for *a*, but if *b* is a descriptor, the respective descriptor method gets called. Understanding descriptors is a key to a deep understanding of Python because they are the basis for many features including functions, methods, properties, class methods, static methods, and reference to super classes.

Pour plus d'informations sur les méthodes des descripteurs, consultez descriptors ou le guide pour l'utilisation des descripteurs.

dictionnaire

An associative array, where arbitrary keys are mapped to values. The keys can be any object with `__hash__()` and `__eq__()` methods. Called a hash in Perl.

dictionnaire en compréhension (ou dictionnaire en intension)

Écriture concise pour traiter tout ou partie des éléments d'un itérable et renvoyer un dictionnaire contenant les résultats. `results = {n: n ** 2 for n in range(10)}` génère un dictionnaire contenant des clés `n` liées à leurs valeurs `n ** 2`. Voir compréhensions.

vue de dictionnaire

Objets retournés par les méthodes `dict.keys()`, `dict.values()` et `dict.items()`. Ils fournissent des vues dynamiques des entrées du dictionnaire, ce qui signifie que lorsque le dictionnaire change, la vue change. Pour transformer une vue en vraie liste, utilisez `list(dictview)`. Voir *Les vues de dictionnaires*.

chaîne de documentation (*docstring*)

A string literal which appears as the first expression in a class, function or module. While ignored when the suite is executed, it is recognized by the compiler and put into the `__doc__` attribute of the enclosing class, function or module. Since it is available via introspection, it is the canonical place for documentation of the object.

typage canard (*duck-typing*)

Style de programmation qui ne prend pas en compte le type d'un objet pour déterminer s'il respecte une interface, mais qui appelle simplement la méthode ou l'attribut (*Si ça a un bec et que ça cancane, ça doit être un canard*, *duck* signifie canard en anglais). En se concentrant sur les interfaces plutôt que les types, du code bien construit améliore sa flexibilité en autorisant des substitutions polymorphiques. Le *duck-typing* évite de vérifier les types via `type()` ou `isinstance()`, Notez cependant que le *duck-typing* peut travailler de pair avec les *classes mère abstraites*. À la place, le *duck-typing* utilise plutôt `hasattr()` ou la programmation *EAFP*.

EAFP

Il est plus simple de demander pardon que demander la permission (*Easier to Ask for Forgiveness than Permission* en anglais). Ce style de développement Python fait l'hypothèse que le code est valide et traite les exceptions si cette hypothèse s'avère fausse. Ce style, propre et efficace, est caractérisé par la présence de beaucoup de mots clés `try` et `except`. Cette technique de programmation contraste avec le style *LBYL* utilisé couramment dans les langages tels que C.

evaluate function
A function that can be called to evaluate a lazily evaluated attribute of an object, such as the value of type aliases created with the `type` statement.

expression
Suite logique de termes et chiffres conformes à la syntaxe Python dont l'évaluation fournit une valeur. En d'autres termes, une expression est une suite d'éléments tels que des noms, opérateurs, littéraux, accès d'attributs, méthodes ou fonctions qui aboutissent à une valeur. Contrairement à beaucoup d'autres langages, les différentes constructions du langage ne sont pas toutes des expressions. On trouve également des *instructions* qui ne peuvent pas être utilisées comme expressions, tel que `while`. Les affectations sont également des instructions et non des expressions.

module d'extension
Module écrit en C ou C++, utilisant l'API C de Python pour interagir avec Python et le code de l'utilisateur.

f-string
Chaîne littérale préfixée de `'f'` ou `'F'`. Les "f-strings" sont un raccourci pour formatted string literals. Voir la **PEP 498**.

objet fichier
An object exposing a file-oriented API (with methods such as `read()` or `write()`) to an underlying resource. Depending on the way it was created, a file object can mediate access to a real on-disk file or to another type of storage or communication device (for example standard input/output, in-memory buffers, sockets, pipes, etc.). File objects are also called *file-like objects* or *streams*.

Il existe en réalité trois catégories de fichiers objets : les *fichiers binaires* bruts, les *fichiers binaires* avec tampon (*buffer*) et les *fichiers textes*. Leurs interfaces sont définies dans le module `io`. Le moyen le plus simple et direct de créer un objet fichier est d'utiliser la fonction `open()`.

objet fichier-compatible
Synonyme de *objet fichier*.

encodage du système de fichiers et gestionnaire d'erreurs associé
Encodage et gestionnaire d'erreurs utilisés par Python pour décoder les octets fournis par le système d'exploitation et encoder les chaînes de caractères Unicode afin de les passer au système.

L'encodage du système de fichiers doit impérativement pouvoir décoder tous les octets jusqu'à 128. Si ce n'est pas le cas, certaines fonctions de l'API lèvent `UnicodeError`.

Cet encodage et son gestionnaire d'erreur peuvent être obtenus à l'aide des fonctions `sys.getfilesystemencoding()` et `sys.getfilesystemencodeerrors()`.

L'*encodage du système de fichiers et gestionnaire d'erreurs associé* sont configurés au démarrage de Python par la fonction `PyConfig_Read()` : regardez `filesystem_encoding` et `filesystem_errors` dans les membres de `PyConfig`.

Voir aussi *encodage régional*.

chercheur
Objet qui essaie de trouver un *chargeur* pour le module en cours d'importation.

There are two types of finder : *meta path finders* for use with `sys.meta_path`, and *path entry finders* for use with `sys.path_hooks`.

See finders-and-loaders and `importlib` for much more detail.

division entière
Division mathématique arrondissant à l'entier inférieur. L'opérateur de la division entière est `//`. Par exemple l'expression `11 // 4` vaut `2`, contrairement à `11 / 4` qui vaut `2.75`. Notez que `(-11) // 4` vaut `-3` car l'arrondi se fait à l'entier inférieur. Voir la **PEP 328**.

free threading
A threading model where multiple threads can run Python bytecode simultaneously within the same interpreter. This is in contrast to the *global interpreter lock* which allows only one thread to execute Python bytecode at a time. See **PEP 703**.

free variable
Formally, as defined in the language execution model, a free variable is any variable used in a namespace which is not a local variable in that namespace. See *closure variable* for an example. Pragmatically, due to the name of the `codeobject.co_freevars` attribute, the term is also sometimes used as a synonym for *closure variable*.

fonction
Suite d'instructions qui renvoie une valeur à son appelant. On peut lui passer des *arguments* qui pourront être utilisés dans le corps de la fonction. Voir aussi *paramètre*, *méthode* et function.

annotation de fonction
annotation d'un paramètre de fonction ou valeur de retour.

Les annotations de fonctions sont généralement utilisées pour des *indications de types* : par exemple, cette fonction devrait prendre deux arguments `int` et devrait également avoir une valeur de retour de type `int` :

```
def sum_two_numbers(a: int, b: int) -> int:
   return a + b
```

L'annotation syntaxique de la fonction est expliquée dans la section function.

Voir *annotation de variable* et la **PEP 484**, qui décrivent cette fonctionnalité. Voir aussi annotations-howto sur les bonnes pratiques concernant les annotations.

__future__
Une importation depuis le futur s'écrit `from __future__ import <fonctionnalité>`. Lorsqu'une importation du futur est active dans un module, Python compile ce module avec une certaine modification de la syntaxe ou du comportement qui est vouée à devenir standard dans une version ultérieure. Le module `__future__` documente les possibilités pour *fonctionnalité*. L'importation a aussi l'effet normal d'importer une variable du module. Cette variable contient des informations utiles sur la fonctionnalité en question, notamment la version de Python dans laquelle elle a été ajoutée, et celle dans laquelle elle deviendra standard :

```
>>> import __future__
>>> __future__.division
_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)
```

ramasse-miettes
(*garbage collection* en anglais) Mécanisme permettant de libérer de la mémoire lorsqu'elle n'est plus utilisée. Python utilise un ramasse-miettes par comptage de référence et un ramasse-miettes cyclique capable de détecter et casser les références circulaires. Le ramasse-miettes peut être contrôlé en utilisant le module `gc`.

générateur
Fonction qui renvoie un *itérateur de générateur*. Cela ressemble à une fonction normale, en dehors du fait qu'elle contient une ou des expressions `yield` produisant une série de valeurs utilisable dans une boucle *for* ou récupérées une à une via la fonction `next()`.

Fait généralement référence à une fonction génératrice mais peut faire référence à un *itérateur de générateur* dans certains contextes. Dans les cas où le sens voulu n'est pas clair, utiliser les termes complets lève l'ambiguïté.

itérateur de générateur
Objet créé par une fonction *générateur*.

Each `yield` temporarily suspends processing, remembering the execution state (including local variables and pending try-statements). When the *generator iterator* resumes, it picks up where it left off (in contrast to functions which start fresh on every invocation).

expression génératrice
An *expression* that returns an *iterator*. It looks like a normal expression followed by a `for` clause defining a loop variable, range, and an optional `if` clause. The combined expression generates values for an enclosing function :

```
>>> sum(i*i for i in range(10))         # sum of squares 0, 1, 4, ... 81
285
```

fonction générique
Fonction composée de plusieurs fonctions implémentant les mêmes opérations pour différents types. L'implémentation à utiliser est déterminée lors de l'appel par l'algorithme de répartition.

Voir aussi *single dispatch*, le décorateur `functools.singledispatch()` et la **PEP 443**.

type générique
Un *type* qui peut être paramétré ; généralement un conteneur comme `list` ou `dict`. Utilisé pour les *indications de type* et les *annotations*.

Pour plus de détails, voir *types alias génériques* et le module `typing`. On trouvera l'historique de cette fonctionnalité dans les **PEP 483**, **PEP 484** et **PEP 585**.

GIL

Voir *global interpreter lock*.

verrou global de l'interpréteur

(*global interpreter lock* en anglais) Mécanisme utilisé par l'interpréteur *CPython* pour s'assurer qu'un seul fil d'exécution (*thread* en anglais) n'exécute le *bytecode* à la fois. Cela simplifie l'implémentation de CPython en rendant le modèle objet (incluant des parties critiques comme la classe native `dict`) implicitement protégé contre les accès concourants. Verrouiller l'interpréteur entier rend plus facile l'implémentation de multiples fils d'exécution (*multi-thread* en anglais), au détriment malheureusement de beaucoup du parallélisme possible sur les machines ayant plusieurs processeurs.

Cependant, certains modules d'extension, standards ou non, sont conçus de manière à libérer le GIL lorsqu'ils effectuent des tâches lourdes tel que la compression ou le hachage. De la même manière, le GIL est toujours libéré lors des entrées-sorties.

As of Python 3.13, the GIL can be disabled using the `--disable-gil` build configuration. After building Python with this option, code must be run with `-X gil=0` or after setting the `PYTHON_GIL=0` environment variable. This feature enables improved performance for multi-threaded applications and makes it easier to use multi-core CPUs efficiently. For more details, see **PEP 703**.

***pyc* utilisant le hachage**

Un fichier de cache de code intermédiaire (*bytecode* en anglais) qui utilise le hachage plutôt que l'heure de dernière modification du fichier source correspondant pour déterminer sa validité. Voir pyc-invalidation.

hachable

An object is *hashable* if it has a hash value which never changes during its lifetime (it needs a `__hash__()` method), and can be compared to other objects (it needs an `__eq__()` method). Hashable objects which compare equal must have the same hash value.

La hachabilité permet à un objet d'être utilisé comme clé de dictionnaire ou en tant que membre d'un ensemble (type *set*), car ces structures de données utilisent ce *hash*.

La plupart des types immuables natifs de Python sont hachables, mais les conteneurs mutables (comme les listes ou les dictionnaires) ne le sont pas ; les conteneurs immuables (comme les n-uplets ou les ensembles figés) ne sont hachables que si leurs éléments sont hachables. Les instances de classes définies par les utilisateurs sont hachables par défaut. Elles sont toutes considérées différentes (sauf avec elles-mêmes) et leur valeur de hachage est calculée à partir de leur `id()`.

IDLE

Environnement d'apprentissage et de développement intégré pour Python. *IDLE* est un éditeur basique et un interpréteur livré avec la distribution standard de Python.

immortal

Immortal objects are a CPython implementation detail introduced in **PEP 683**.

If an object is immortal, its *reference count* is never modified, and therefore it is never deallocated while the interpreter is running. For example, `True` and `None` are immortal in CPython.

Immortal objects can be identified via `sys._is_immortal()`, or via `PyUnstable_IsImmortal()` in the C API.

immuable

Objet dont la valeur ne change pas. Les nombres, les chaînes et les *n*-uplets sont immuables. Ils ne peuvent être modifiés. Un nouvel objet doit être créé si une valeur différente doit être stockée. Ils jouent un rôle important quand une valeur de *hash* constante est requise, typiquement en clé de dictionnaire.

chemin des importations

Liste de *entrées* dans lesquelles le *chercheur basé sur les chemins* cherche les modules à importer. Typiquement, lors d'une importation, cette liste vient de `sys.path` ; pour les sous-paquets, elle peut aussi venir de l'attribut `__path__` du paquet parent.

importation

Processus rendant le code Python d'un module disponible dans un autre.

importateur

Objet qui trouve et charge un module, en même temps un *chercheur* et un *chargeur*.

interactif

Python has an interactive interpreter which means you can enter statements and expressions at the interpreter prompt, immediately execute them and see their results. Just launch `python` with no arguments (possibly by selecting it from your computer's main menu). It is a very powerful way to test out new ideas or inspect modules and packages (remember `help(x)`). For more on interactive mode, see tut-interac.

interprété

Python est un langage interprété, en opposition aux langages compilés, bien que la frontière soit floue en raison de la présence d'un compilateur en code intermédiaire. Cela signifie que les fichiers sources peuvent être exécutés directement, sans avoir à compiler un fichier exécutable intermédiaire. Les langages interprétés ont généralement un cycle de développement / débogage plus court que les langages compilés. Cependant, ils s'exécutent généralement plus lentement. Voir aussi *interactif*.

arrêt de l'interpréteur

Lorsqu'on lui demande de s'arrêter, l'interpréteur Python entre dans une phase spéciale où il libère graduellement les ressources allouées, comme les modules ou quelques structures de données internes. Il fait aussi quelques appels au *ramasse-miettes*. Cela peut déclencher l'exécution de code dans des destructeurs ou des fonctions de rappels de *weakrefs*. Le code exécuté lors de l'arrêt peut rencontrer des exceptions puisque les ressources auxquelles il fait appel sont susceptibles de ne plus fonctionner, (typiquement les modules des bibliothèques ou le mécanisme de *warning*).

La principale raison d'arrêt de l'interpréteur est que le module `__main__` ou le script en cours d'exécution a terminé de s'exécuter.

itérable

An object capable of returning its members one at a time. Examples of iterables include all sequence types (such as `list`, `str`, and `tuple`) and some non-sequence types like `dict`, *file objects*, and objects of any classes you define with an `__iter__()` method or with a `__getitem__()` method that implements *sequence* semantics.

Iterables can be used in a `for` loop and in many other places where a sequence is needed (`zip()`, `map()`, ...). When an iterable object is passed as an argument to the built-in function `iter()`, it returns an iterator for the object. This iterator is good for one pass over the set of values. When using iterables, it is usually not necessary to call `iter()` or deal with iterator objects yourself. The `for` statement does that automatically for you, creating a temporary unnamed variable to hold the iterator for the duration of the loop. See also *iterator*, *sequence*, and *generator*.

itérateur

An object representing a stream of data. Repeated calls to the iterator's `__next__()` method (or passing it to the built-in function `next()`) return successive items in the stream. When no more data are available a `StopIteration` exception is raised instead. At this point, the iterator object is exhausted and any further calls to its `__next__()` method just raise `StopIteration` again. Iterators are required to have an `__iter__()` method that returns the iterator object itself so every iterator is also iterable and may be used in most places where other iterables are accepted. One notable exception is code which attempts multiple iteration passes. A container object (such as a `list`) produces a fresh new iterator each time you pass it to the `iter()` function or use it in a `for` loop. Attempting this with an iterator will just return the same exhausted iterator object used in the previous iteration pass, making it appear like an empty container.

Vous trouverez davantage d'informations dans *Les types itérateurs*.

Particularité de l'implémentation CPython : CPython does not consistently apply the requirement that an iterator define `__iter__()`. And also please note that the free-threading CPython does not guarantee the thread-safety of iterator operations.

fonction clé

Une fonction clé est un objet appelable qui renvoie une valeur à fins de tri ou de classement. Par exemple, la fonction `locale.strxfrm()` est utilisée pour générer une clé de classement prenant en compte les conventions de classement spécifiques aux paramètres régionaux courants.

Plusieurs outils dans Python acceptent des fonctions clés pour déterminer comment les éléments sont classés ou groupés. On peut citer les fonctions `min()`, `max()`, `sorted()`, `list.sort()`, `heapq.merge()`, `heapq.nsmallest()`, `heapq.nlargest()` et `itertools.groupby()`.

Il existe plusieurs moyens de créer une fonction clé. Par exemple, la méthode `str.lower()` peut servir de fonction clé pour effectuer des recherches insensibles à la casse. Aussi, il est possible de créer des fonctions clés avec des expressions `lambda`, comme `lambda r: (r[0], r[2])`. Par ailleurs `attrgetter()`, `itemgetter()` et `methodcaller()` permettent de créer des fonctions clés. Voir le guide pour le tri pour

des exemples de création et d'utilisation de fonctions clefs.

argument nommé
Voir *argument*.

lambda
Fonction anonyme sous la forme d'une *expression* et ne contenant qu'une seule expression, exécutée lorsque la fonction est appelée. La syntaxe pour créer des fonctions lambda est : `lambda [parameters]: expression`

LBYL
Regarde avant de sauter, (*Look before you leap* en anglais). Ce style de programmation consiste à vérifier des conditions avant d'effectuer des appels ou des accès. Ce style contraste avec le style *EAFP* et se caractérise par la présence de beaucoup d'instructions `if`.

Dans un environnement avec plusieurs fils d'exécution (*multi-threaded* en anglais), le style *LBYL* peut engendrer un séquencement critique (*race condition* en anglais) entre le "regarde" et le "sauter". Par exemple, le code `if key in mapping: return mapping[key]` peut échouer si un autre fil d'exécution supprime la clé *key* du *mapping* après le test mais avant l'accès. Ce problème peut être résolu avec des verrous (*locks*) ou avec l'approche EAFP.

liste
A built-in Python *sequence*. Despite its name it is more akin to an array in other languages than to a linked list since access to elements is $O(1)$.

liste en compréhension (ou liste en intension)
Écriture concise pour manipuler tout ou partie des éléments d'une séquence et renvoyer une liste contenant les résultats. `result = ['{:#04x}'.format(x) for x in range(256) if x % 2 == 0]` génère la liste composée des nombres pairs de 0 à 255 écrits sous formes de chaînes de caractères et en hexadécimal (`0x`...). La clause `if` est optionnelle. Si elle est omise, tous les éléments du `range(256)` seront utilisés.

chargeur
An object that loads a module. It must define the `exec_module()` and `create_module()` methods to implement the `Loader` interface. A loader is typically returned by a *finder*. See also :
— finders-and-loaders
— `importlib.abc.Loader`
— **PEP 302**

encodage régional
Sous Unix, il est défini par la variable régionale LC_CTYPE. Il peut être modifié par `locale.setlocale(locale.LC_CTYPE, new_locale)`.

Sous Windows, c'est un encodage ANSI (par ex. : `"cp1252"`).

Sous Android et VxWorks, Python utilise `"utf-8"` comme encodage régional.

`locale.getencoding()` can be used to get the locale encoding.

Voir aussi l'*encodage du systèmes de fichiers et gestionnaire d'erreurs associé*.

méthode magique
Un synonyme informel de *special method*.

tableau de correspondances (*mapping* en anglais)
Conteneur permettant de rechercher des éléments à partir de clés et implémentant les méthodes spécifiées dans les classes mères abstraites des `tableaux de correspondances` (immuables) ou `tableaux de correspondances mutables` (voir les *classes mères abstraites*). Les classes suivantes sont des exemples de tableaux de correspondances : `dict`, `collections.defaultdict`, `collections.OrderedDict` et `collections.Counter`.

chercheur dans les méta-chemins
Un *chercheur* renvoyé par une recherche dans `sys.meta_path`. Les chercheurs dans les méta-chemins ressemblent, mais sont différents des *chercheurs d'entrée dans path*.

Voir `importlib.abc.MetaPathFinder` pour les méthodes que les chercheurs dans les méta-chemins doivent implémenter.

métaclasse
Classe d'une classe. Les définitions de classe créent un nom pour la classe, un dictionnaire de classe et une liste de classes parentes. La métaclasse a pour rôle de réunir ces trois paramètres pour construire la classe. La plupart des langages orientés objet fournissent une implémentation par défaut. La particularité de Python

est la possibilité de créer des métaclasses personnalisées. La plupart des utilisateurs n'auront jamais besoin de cet outil, mais lorsque le besoin survient, les métaclasses offrent des solutions élégantes et puissantes. Elles sont utilisées pour journaliser les accès à des propriétés, rendre sûrs les environnements *multi-threads*, suivre la création d'objets, implémenter des singletons et bien d'autres tâches.

Plus d'informations sont disponibles dans : metaclasses.

méthode
Fonction définie à l'intérieur d'une classe. Lorsqu'elle est appelée comme un attribut d'une instance de cette classe, la méthode reçoit l'instance en premier *argument* (qui, par convention, est habituellement nommé `self`). Voir *function* et *nested scope*.

ordre de résolution des méthodes
Method Resolution Order is the order in which base classes are searched for a member during lookup. See python_2.3_mro for details of the algorithm used by the Python interpreter since the 2.3 release.

module
Objet utilisé pour organiser une portion unitaire de code en Python. Les modules ont un espace de nommage et peuvent contenir n'importe quels objets Python. Charger des modules est appelé *importer*.

Voir aussi *paquet*.

spécificateur de module
Espace de nommage contenant les informations, relatives à l'importation, utilisées pour charger un module. C'est une instance de la classe `importlib.machinery.ModuleSpec`.

See also module-specs.

MRO
Voir *ordre de résolution des méthodes*.

mutable
Un objet mutable peut changer de valeur tout en gardant le même `id()`. Voir aussi *immuable*.

***n*-uplet nommé**
Le terme "n-uplet nommé" s'applique à tous les types ou classes qui héritent de la classe `tuple` et dont les éléments indexables sont aussi accessibles en utilisant des attributs nommés. Les types et classes peuvent avoir aussi d'autres caractéristiques.

Plusieurs types natifs sont appelés n-uplets, y compris les valeurs retournées par `time.localtime()` et `os.stat()`. Un autre exemple est `sys.float_info` :

```
>>> sys.float_info[1]                    # indexed access
1024
>>> sys.float_info.max_exp               # named field access
1024
>>> isinstance(sys.float_info, tuple)    # kind of tuple
True
```

Some named tuples are built-in types (such as the above examples). Alternatively, a named tuple can be created from a regular class definition that inherits from `tuple` and that defines named fields. Such a class can be written by hand, or it can be created by inheriting `typing.NamedTuple`, or with the factory function `collections.namedtuple()`. The latter techniques also add some extra methods that may not be found in hand-written or built-in named tuples.

espace de nommage
L'endroit où une variable est stockée. Les espaces de nommage sont implémentés avec des dictionnaires. Il existe des espaces de nommage globaux, natifs ou imbriqués dans les objets (dans les méthodes). Les espaces de nommage favorisent la modularité car ils permettent d'éviter les conflits de noms. Par exemple, les fonctions `builtins.open` et `os.open()` sont différenciées par leurs espaces de nom. Les espaces de nommage aident aussi à la lisibilité et la maintenabilité en rendant clair quel module implémente une fonction. Par exemple, écrire `random.seed()` ou `itertools.islice()` affiche clairement que ces fonctions sont implémentées respectivement dans les modules `random` et `itertools`.

paquet-espace de nommage
A *package* which serves only as a container for subpackages. Namespace packages may have no physical representation, and specifically are not like a *regular package* because they have no `__init__.py` file.

Namespace packages allow several individually installable packages to have a common parent package. Other-

wise, it is recommended to use a *regular package*.

For more information, see **PEP 420** and reference-namespace-package.

Voir aussi *module*.

portée imbriquée
Possibilité de faire référence à une variable déclarée dans une définition englobante. Typiquement, une fonction définie à l'intérieur d'une autre fonction a accès aux variables de cette dernière. Souvenez-vous cependant que cela ne fonctionne que pour accéder à des variables, pas pour les assigner. Les variables locales sont lues et assignées dans l'espace de nommage le plus proche. Tout comme les variables globales qui sont stockés dans l'espace de nommage global, le mot clef `nonlocal` permet d'écrire dans l'espace de nommage dans lequel est déclarée la variable.

nouvelle classe
Old name for the flavor of classes now used for all class objects. In earlier Python versions, only new-style classes could use Python's newer, versatile features like `__slots__`, descriptors, properties, `__getattribute__()`, class methods, and static methods.

objet
N'importe quelle donnée comportant des états (sous forme d'attributs ou d'une valeur) et un comportement (des méthodes). C'est aussi (`object`) l'ancêtre commun à absolument toutes les *nouvelles classes*.

optimized scope
A scope where target local variable names are reliably known to the compiler when the code is compiled, allowing optimization of read and write access to these names. The local namespaces for functions, generators, coroutines, comprehensions, and generator expressions are optimized in this fashion. Note : most interpreter optimizations are applied to all scopes, only those relying on a known set of local and nonlocal variable names are restricted to optimized scopes.

paquet
module Python qui peut contenir des sous-modules ou des sous-paquets. Techniquement, un paquet est un module qui possède un attribut `__path__`.

Voir aussi *paquet classique* et *namespace package*.

paramètre
Entité nommée dans la définition d'une *fonction* (ou méthode), décrivant un *argument* (ou dans certains cas des arguments) que la fonction accepte. Il existe cinq sortes de paramètres :

— *positional-or-keyword* : l'argument peut être passé soit par sa *position*, soit en tant que *argument nommé*. C'est le type de paramètre par défaut. Par exemple, *foo* et *bar* dans l'exemple suivant :

```
def func(foo, bar=None): ...
```

— *positional-only* : définit un argument qui ne peut être fourni que par position. Les paramètres *positional-only* peuvent être définis en insérant un caractère "/" dans la liste de paramètres de la définition de fonction après eux. Par exemple : *posonly1* et *posonly2* dans le code suivant :

```
def func(posonly1, posonly2, /, positional_or_keyword): ...
```

— *keyword-only* : l'argument ne peut être fourni que nommé. Les paramètres *keyword-only* peuvent être définis en utilisant un seul paramètre *var-positional*, ou en ajoutant une étoile (`*`) seule dans la liste des paramètres avant eux. Par exemple, *kw_only1* et *kw_only2* dans le code suivant :

```
def func(arg, *, kw_only1, kw_only2): ...
```

— *var-positional* : une séquence d'arguments positionnels peut être fournie (en plus de tous les arguments positionnels déjà acceptés par d'autres paramètres). Un tel paramètre peut être défini en préfixant son nom par une `*`. Par exemple *args* ci-après :

```
def func(*args, **kwargs): ...
```

— *var-keyword* : une quantité arbitraire d'arguments peut être passée, chacun étant nommé (en plus de tous les arguments nommés déjà acceptés par d'autres paramètres). Un tel paramètre est défini en préfixant le nom du paramètre par `**`. Par exemple, *kwargs* ci-dessus.

Les paramètres peuvent spécifier des arguments obligatoires ou optionnels, ainsi que des valeurs par défaut pour les arguments optionnels.

Voir aussi *argument* dans le glossaire, la question sur la différence entre les arguments et les paramètres dans la FAQ, la classe `inspect.Parameter`, la section function et la **PEP 362**.

entrée de chemin
Emplacement dans le *chemin des importations* (*import path* en anglais, d'où le *path*) que le *chercheur basé sur les chemins* consulte pour trouver des modules à importer.

chercheur de chemins
chercheur renvoyé par un appelable sur un `sys.path_hooks` (c'est-à-dire un *point d'entrée pour la recherche dans path*) qui sait où trouver des modules lorsqu'on lui donne une *entrée de path*.

Voir `importlib.abc.PathEntryFinder` pour les méthodes qu'un chercheur d'entrée dans *path* doit implémenter.

point d'entrée pour la recherche dans *path*
A callable on the `sys.path_hooks` list which returns a *path entry finder* if it knows how to find modules on a specific *path entry*.

chercheur basé sur les chemins
L'un des *chercheurs dans les méta-chemins* par défaut qui cherche des modules dans un *chemin des importations*.

objet simili-chemin
Objet représentant un chemin du système de fichiers. Un objet simili-chemin est un objet `str` ou un objet `bytes` représentant un chemin ou un objet implémentant le protocole `os.PathLike`. Un objet qui accepte le protocole `os.PathLike` peut être converti en un chemin `str` ou `bytes` du système de fichiers en appelant la fonction `os.fspath()`. `os.fsdecode()` et `os.fsencode()` peuvent être utilisées, respectivement, pour garantir un résultat de type `str` ou `bytes` à la place. A été Introduit par la **PEP 519**.

PEP
Python Enhancement Proposal (Proposition d'amélioration de Python). Une PEP est un document de conception fournissant des informations à la communauté Python ou décrivant une nouvelle fonctionnalité pour Python, ses processus ou son environnement. Les PEP doivent fournir une spécification technique concise et une justification des fonctionnalités proposées.

Les PEP sont censées être les principaux mécanismes pour proposer de nouvelles fonctionnalités majeures, pour recueillir les commentaires de la communauté sur une question et pour documenter les décisions de conception qui sont intégrées en Python. L'auteur du PEP est responsable de l'établissement d'un consensus au sein de la communauté et de documenter les opinions contradictoires.

Voir la **PEP 1**.

portion
Jeu de fichiers dans un seul dossier (pouvant être stocké sous forme de fichier zip) qui contribue à l'espace de nommage d'un paquet, tel que défini dans la **PEP 420**.

argument positionnel
Voir *argument*.

API provisoire
Une API provisoire est une API qui n'offre aucune garantie de rétrocompatibilité (la bibliothèque standard exige la rétrocompatibilité). Bien que des changements majeurs d'une telle interface ne soient pas attendus, tant qu'elle est étiquetée provisoire, des changements cassant la rétrocompatibilité (y compris sa suppression complète) peuvent survenir si les développeurs principaux le jugent nécessaire. Ces modifications ne surviendront que si de sérieux problèmes sont découverts et qu'ils n'avaient pas été identifiés avant l'ajout de l'API.

Même pour les API provisoires, les changements cassant la rétrocompatibilité sont considérés comme des "solutions de dernier recours". Tout ce qui est possible sera fait pour tenter de résoudre les problèmes en conservant la rétrocompatibilité.

Ce processus permet à la bibliothèque standard de continuer à évoluer avec le temps, sans se bloquer longtemps sur des erreurs d'architecture. Voir la **PEP 411** pour plus de détails.

paquet provisoire
Voir *provisional API*.

Python 3000
Surnom donné à la série des Python 3.x (très vieux surnom donné à l'époque où Python 3 représentait un futur lointain). Aussi abrégé *Py3k*.

Pythonique

Idée, ou bout de code, qui colle aux idiomes de Python plutôt qu'aux concepts communs rencontrés dans d'autres langages. Par exemple, il est idiomatique en Python de parcourir les éléments d'un itérable en utilisant `for`. Beaucoup d'autres langages n'ont pas cette possibilité, donc les gens qui ne sont pas habitués à Python utilisent parfois un compteur numérique à la place :

```
for i in range(len(food)):
    print(food[i])
```

Plutôt qu'utiliser la méthode, plus propre et élégante, donc *Pythonique* :

```
for piece in food:
    print(piece)
```

nom qualifié

Nom, comprenant des points, montrant le "chemin" de l'espace de nommage global d'un module vers une classe, fonction ou méthode définie dans ce module, tel que défini dans la **PEP 3155**. Pour les fonctions et classes de premier niveau, le nom qualifié est le même que le nom de l'objet :

```
>>> class C:
...     class D:
...         def meth(self):
...             pass
...
>>> C.__qualname__
'C'
>>> C.D.__qualname__
'C.D'
>>> C.D.meth.__qualname__
'C.D.meth'
```

Lorsqu'il est utilisé pour nommer des modules, le *nom qualifié complet* (*fully qualified name* - *FQN* en anglais) signifie le chemin complet (séparé par des points) vers le module, incluant tous les paquets parents. Par exemple : `email.mime.text` :

```
>>> import email.mime.text
>>> email.mime.text.__name__
'email.mime.text'
```

nombre de références

The number of references to an object. When the reference count of an object drops to zero, it is deallocated. Some objects are *immortal* and have reference counts that are never modified, and therefore the objects are never deallocated. Reference counting is generally not visible to Python code, but it is a key element of the *CPython* implementation. Programmers can call the `sys.getrefcount()` function to return the reference count for a particular object.

paquet classique

paquet traditionnel, tel qu'un dossier contenant un fichier `__init__.py`.

Voir aussi *paquet-espace de nommage*.

REPL

An acronym for the "read–eval–print loop", another name for the *interactive* interpreter shell.

__slots__

Déclaration dans une classe qui économise de la mémoire en pré-allouant de l'espace pour les attributs des instances et qui élimine le dictionnaire (des attributs) des instances. Bien que populaire, cette technique est difficile à maîtriser et devrait être réservée à de rares cas où un grand nombre d'instances dans une application devient un sujet critique pour la mémoire.

séquence

An *iterable* which supports efficient element access using integer indices via the `__getitem__()` special method and defines a `__len__()` method that returns the length of the sequence. Some built-in sequence types are `list`, `str`, `tuple`, and `bytes`. Note that `dict` also supports `__getitem__()` and `__len__()`,

but is considered a mapping rather than a sequence because the lookups use arbitrary *hashable* keys rather than integers.

The `collections.abc.Sequence` abstract base class defines a much richer interface that goes beyond just `__getitem__()` and `__len__()`, adding `count()`, `index()`, `__contains__()`, and `__reversed__()`. Types that implement this expanded interface can be registered explicitly using `register()`. For more documentation on sequence methods generally, see *Common Sequence Operations*.

ensemble en compréhension (ou ensemble en intension)
Une façon compacte de traiter tout ou partie des éléments d'un itérable et de renvoyer un *set* avec les résultats. `results = {c for c in 'abracadabra' if c not in 'abc'}` génère l'ensemble contenant les lettres « r » et « d » `{'r', 'd'}`. Voir comprehensions.

distribution simple
Forme de distribution, comme les *fonction génériques*, où l'implémentation est choisie en fonction du type d'un seul argument.

tranche
(*slice* en anglais), un objet contenant habituellement une portion de *séquence*. Une tranche est créée en utilisant la notation `[]` avec des `:` entre les nombres lorsque plusieurs sont fournis, comme dans `variable_name[1:3:5]`. Cette notation utilise des objets `slice` en interne.

soft deprecated
A soft deprecated API should not be used in new code, but it is safe for already existing code to use it. The API remains documented and tested, but will not be enhanced further.

Soft deprecation, unlike normal deprecation, does not plan on removing the API and will not emit warnings.

See PEP 387 : Soft Deprecation.

méthode spéciale
(*special method* en anglais) Méthode appelée implicitement par Python pour exécuter une opération sur un type, comme une addition. De telles méthodes ont des noms commençant et terminant par des doubles tirets bas. Les méthodes spéciales sont documentées dans specialnames.

instruction
Une instruction (*statement* en anglais) est un composant d'un "bloc" de code. Une instruction est soit une *expression*, soit une ou plusieurs constructions basées sur un mot-clé, comme `if`, `while` ou `for`.

static type checker
An external tool that reads Python code and analyzes it, looking for issues such as incorrect types. See also *type hints* and the `typing` module.

référence forte
In Python's C API, a strong reference is a reference to an object which is owned by the code holding the reference. The strong reference is taken by calling `Py_INCREF()` when the reference is created and released with `Py_DECREF()` when the reference is deleted.

Une référence forte est créée à l'aide de la fonction `Py_NewRef()`. Il faut normalement appeler `Py_DECREF()` dessus avant de sortir de sa portée lexicale, sans quoi il y a une fuite de référence.

Voir aussi *référence empruntée*.

encodages de texte
Une chaîne de caractères en Python est une suite de points de code Unicode (dans l'intervalle `U+0000`--`U+10FFFF`). Pour stocker ou transmettre une chaîne, il est nécessaire de la sérialiser en suite d'octets.

Sérialiser une chaîne de caractères en une suite d'octets s'appelle « encoder » et recréer la chaîne à partir de la suite d'octets s'appelle « décoder ».

Il existe de multiples *codecs* pour la sérialisation de texte, que l'on regroupe sous l'expression « encodages de texte ».

fichier texte
Objet fichier capable de lire et d'écrire des objets `str`. Souvent, un fichier texte (*text file* en anglais) accède en fait à un flux de donnée en octets et gère l'*encodage de texte* automatiquement. Des exemples de fichiers textes sont les fichiers ouverts en mode texte (`'r'` ou `'w'`), `sys.stdin`, `sys.stdout` et les instances de `io.StringIO`.

Voir aussi *fichier binaire* pour un objet fichier capable de lire et d'écrire des *objets octets-compatibles*.

chaîne entre triple guillemets

Chaîne qui est délimitée par trois guillemets simples (`'`) ou trois guillemets doubles (`"`). Bien qu'elle ne fournisse aucune fonctionnalité qui ne soit pas disponible avec une chaîne entre guillemets, elle est utile pour de nombreuses raisons. Elle vous autorise à insérer des guillemets simples et doubles dans une chaîne sans avoir à les protéger et elle peut s'étendre sur plusieurs lignes sans avoir à terminer chaque ligne par un `\`. Elle est ainsi particulièrement utile pour les chaînes de documentation (*docstrings*).

type

The type of a Python object determines what kind of object it is ; every object has a type. An object's type is accessible as its `__class__` attribute or can be retrieved with `type(obj)`.

alias de type

Synonyme d'un type, créé en affectant le type à un identifiant.

Les alias de types sont utiles pour simplifier les *indications de types*. Par exemple :

```
def remove_gray_shades(
        colors: list[tuple[int, int, int]]) -> list[tuple[int, int, int]]:
    pass
```

pourrait être rendu plus lisible comme ceci :

```
Color = tuple[int, int, int]

def remove_gray_shades(colors: list[Color]) -> list[Color]:
    pass
```

Voir `typing` et la **PEP 484**, qui décrivent cette fonctionnalité.

indication de type

L'*annotation* qui spécifie le type attendu pour une variable, un attribut de classe, un paramètre de fonction ou une valeur de retour.

Type hints are optional and are not enforced by Python but they are useful to *static type checkers*. They can also aid IDEs with code completion and refactoring.

Les indications de type de variables globales, d'attributs de classe et de fonctions, mais pas de variables locales, peuvent être consultées en utilisant `typing.get_type_hints()`.

Voir `typing` et la **PEP 484**, qui décrivent cette fonctionnalité.

retours à la ligne universels

Une manière d'interpréter des flux de texte dans lesquels sont reconnues toutes les fins de ligne suivantes : la convention Unix `'\n'`, la convention Windows `'\r\n'` et l'ancienne convention Macintosh `'\r'`. Voir la **PEP 278** et la **PEP 3116**, ainsi que la fonction `bytes.splitlines()` pour d'autres usages.

annotation de variable

annotation d'une variable ou d'un attribut de classe.

Lorsque vous annotez une variable ou un attribut de classe, l'affectation est facultative :

```
class C:
    field: 'annotation'
```

Les annotations de variables sont généralement utilisées pour des *indications de types* : par exemple, cette variable devrait prendre des valeurs de type `int` :

```
count: int = 0
```

La syntaxe d'annotation de variable est expliquée dans la section annassign.

Reportez-vous à *annotation de fonction*, à la **PEP 484** et à la **PEP 526** qui décrivent cette fonctionnalité. Voir aussi annotations-howto sur les bonnes pratiques concernant les annotations.

environnement virtuel

Environnement d'exécution isolé (en mode coopératif) qui permet aux utilisateurs de Python et aux applications d'installer et de mettre à jour des paquets sans interférer avec d'autres applications Python fonctionnant sur le même système.

Voir aussi `venv`.

machine virtuelle

Ordinateur défini entièrement par du logiciel. La machine virtuelle (*virtual machine*) de Python exécute le *code intermédiaire* produit par le compilateur de *bytecode*.

Le zen de Python

Liste de principes et de préceptes utiles pour comprendre et utiliser le langage. Cette liste peut être obtenue en tapant "`import this`" dans une invite Python interactive.

ANNEXE B

About this documentation

Python's documentation is generated from reStructuredText sources using Sphinx, a documentation generator originally created for Python and now maintained as an independent project.

Le développement de la documentation et de ses outils est entièrement basé sur le volontariat, tout comme Python. Si vous voulez contribuer, allez voir la page reporting-bugs qui contient des informations pour vous y aider. Les nouveaux volontaires sont toujours les bienvenus !

Merci beaucoup à :

- Fred L. Drake, Jr., the creator of the original Python documentation toolset and author of much of the content ;
- le projet Docutils pour avoir créé *reStructuredText* et la suite d'outils *Docutils* ;
- Fredrik Lundh pour son projet *Alternative Python Reference*, dont Sphinx a pris beaucoup de bonnes idées.

B.1 Contributors to the Python documentation

De nombreuses personnes ont contribué au langage Python, à sa bibliothèque standard et à sa documentation. Consultez Misc/ACKS dans les sources de la distribution Python pour avoir une liste partielle des contributeurs.

Ce n'est que grâce aux suggestions et contributions de la communauté Python que Python a une documentation si merveilleuse — Merci !

ANNEXE C

Histoire et licence

C.1 Histoire du logiciel

Python was created in the early 1990s by Guido van Rossum at Stichting Mathematisch Centrum (CWI, see https://www.cwi.nl) in the Netherlands as a successor of a language called ABC. Guido remains Python's principal author, although it includes many contributions from others.

In 1995, Guido continued his work on Python at the Corporation for National Research Initiatives (CNRI, see https://www.cnri.reston.va.us) in Reston, Virginia where he released several versions of the software.

In May 2000, Guido and the Python core development team moved to BeOpen.com to form the BeOpen PythonLabs team. In October of the same year, the PythonLabs team moved to Digital Creations, which became Zope Corporation. In 2001, the Python Software Foundation (PSF, see https://www.python.org/psf/) was formed, a non-profit organization created specifically to own Python-related Intellectual Property. Zope Corporation was a sponsoring member of the PSF.

All Python releases are Open Source (see https://opensource.org for the Open Source Definition). Historically, most, but not all, Python releases have also been GPL-compatible ; the table below summarizes the various releases.

Version	Dérivé de	Année	Propriétaire	GPL-compatible ? (1)
0.9.0 à 1.2	n/a	1991-1995	CWI	oui
1.3 à 1.5.2	1.2	1995-1999	CNRI	oui
1.6	1.5.2	2000	CNRI	non
2.0	1.6	2000	BeOpen.com	non
1.6.1	1.6	2001	CNRI	yes (2)
2.1	2.0+1.6.1	2001	PSF	non
2.0.1	2.0+1.6.1	2001	PSF	oui
2.1.1	2.1+2.0.1	2001	PSF	oui
2.1.2	2.1.1	2002	PSF	oui
2.1.3	2.1.2	2002	PSF	oui
2.2 et ultérieure	2.1.1	2001-maintenant	PSF	oui

Note

(1) GPL-compatible doesn't mean that we're distributing Python under the GPL. All Python licenses, unlike the GPL, let you distribute a modified version without making your changes open source. The GPL-compatible licenses make it possible to combine Python with other software that is released under the GPL ; the others don't.

(2) According to Richard Stallman, 1.6.1 is not GPL-compatible, because its license has a choice of law clause. According to CNRI, however, Stallman's lawyer has told CNRI's lawyer that 1.6.1 is "not incompatible" with the GPL.

Merci aux nombreux bénévoles qui ont travaillé sous la direction de Guido pour rendre ces versions possibles.

C.2 Conditions générales pour accéder à, ou utiliser, Python

Python software and documentation are licensed under the Python Software Foundation License Version 2.

Starting with Python 3.8.6, examples, recipes, and other code in the documentation are dual licensed under the PSF License Version 2 and the *Zero-Clause BSD license*.

Certains logiciels faisant partie de Python sont soumis à d'autres licences. Ces licences sont incluses avec le code lié à celles-ci. Voir *Licences et remerciements pour les logiciels tiers* pour une liste non exhaustive de ces licences.

C.2.1 PYTHON SOFTWARE FOUNDATION LICENSE VERSION 2

```
1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"), and
   the Individual or Organization ("Licensee") accessing and otherwise using this
   software ("Python") in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, PSF hereby
   grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce,
   analyze, test, perform and/or display publicly, prepare derivative works,
   distribute, and otherwise use Python alone or in any derivative
   version, provided, however, that PSF's License Agreement and PSF's notice of
   copyright, i.e., "Copyright © 2001 Python Software Foundation; All Rights
   Reserved" are retained in Python alone or in any derivative version
   prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on or
   incorporates Python or any part thereof, and wants to make the
   derivative work available to others as provided herein, then Licensee hereby
   agrees to include in any such work a brief summary of the changes made to␣
↪Python.

4. PSF is making Python available to Licensee on an "AS IS" basis.
   PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF
   EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION OR
   WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE
   USE OF PYTHON WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON
   FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON, OR ANY DERIVATIVE
   THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of
   its terms and conditions.

7. Nothing in this License Agreement shall be deemed to create any relationship
```

(suite sur la page suivante)

(suite de la page précédente)

```
   of agency, partnership, or joint venture between PSF and Licensee.  This License
   Agreement does not grant permission to use PSF trademarks or trade name in a
   trademark sense to endorse or promote products or services of Licensee, or any
   third party.

8. By copying, installing or otherwise using Python, Licensee agrees
   to be bound by the terms and conditions of this License Agreement.
```

C.2.2 LICENCE D'UTILISATION BEOPEN.COM POUR PYTHON 2.0

LICENCE D'UTILISATION LIBRE BEOPEN PYTHON VERSION 1

```
1. This LICENSE AGREEMENT is between BeOpen.com ("BeOpen"), having an office at
   160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization
   ("Licensee") accessing and otherwise using this software in source or binary
   form and its associated documentation ("the Software").

2. Subject to the terms and conditions of this BeOpen Python License Agreement,
   BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license
   to reproduce, analyze, test, perform and/or display publicly, prepare derivative
   works, distribute, and otherwise use the Software alone or in any derivative
   version, provided, however, that the BeOpen Python License is retained in the
   Software, alone or in any derivative version prepared by Licensee.

3. BeOpen is making the Software available to Licensee on an "AS IS" basis.
   BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF
   EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR
   WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE
   USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR
   ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING,
   MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF
   ADVISED OF THE POSSIBILITY THEREOF.

5. This License Agreement will automatically terminate upon a material breach of
   its terms and conditions.

6. This License Agreement shall be governed by and interpreted in all respects
   by the law of the State of California, excluding conflict of law provisions.
   Nothing in this License Agreement shall be deemed to create any relationship of
   agency, partnership, or joint venture between BeOpen and Licensee.  This License
   Agreement does not grant permission to use BeOpen trademarks or trade names in a
   trademark sense to endorse or promote products or services of Licensee, or any
   third party.  As an exception, the "BeOpen Python" logos available at
   http://www.pythonlabs.com/logos.html may be used according to the permissions
   granted on that web page.

7. By copying, installing or otherwise using the software, Licensee agrees to be
   bound by the terms and conditions of this License Agreement.
```

C.2.3 LICENCE D'UTILISATION CNRI POUR PYTHON 1.6.1

```
1. This LICENSE AGREEMENT is between the Corporation for National Research
   Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191
```

(suite sur la page suivante)

(suite de la page précédente)

```
   ("CNRI"), and the Individual or Organization ("Licensee") accessing and
   otherwise using Python 1.6.1 software in source or binary form and its
   associated documentation.

2. Subject to the terms and conditions of this License Agreement, CNRI hereby
   grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce,
   analyze, test, perform and/or display publicly, prepare derivative works,
   distribute, and otherwise use Python 1.6.1 alone or in any derivative version,
   provided, however, that CNRI's License Agreement and CNRI's notice of copyright,
   i.e., "Copyright © 1995-2001 Corporation for National Research Initiatives; All
   Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version
   prepared by Licensee.  Alternately, in lieu of CNRI's License Agreement,
   Licensee may substitute the following text (omitting the quotes): "Python 1.6.1
   is made available subject to the terms and conditions in CNRI's License
   Agreement.  This Agreement together with Python 1.6.1 may be located on the
   internet using the following unique, persistent identifier (known as a handle):
   1895.22/1013.  This Agreement may also be obtained from a proxy server on the
   internet using the following URL: http://hdl.handle.net/1895.22/1013".

3. In the event Licensee prepares a derivative work that is based on or
   incorporates Python 1.6.1 or any part thereof, and wants to make the derivative
   work available to others as provided herein, then Licensee hereby agrees to
   include in any such work a brief summary of the changes made to Python 1.6.1.

4. CNRI is making Python 1.6.1 available to Licensee on an "AS IS" basis.  CNRI
   MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF EXAMPLE,
   BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY
   OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF
   PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR
   ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE
   THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of
   its terms and conditions.

7. This License Agreement shall be governed by the federal intellectual property
   law of the United States, including without limitation the federal copyright
   law, and, to the extent such U.S. federal law does not apply, by the law of the
   Commonwealth of Virginia, excluding Virginia's conflict of law provisions.
   Notwithstanding the foregoing, with regard to derivative works based on Python
   1.6.1 that incorporate non-separable material that was previously distributed
   under the GNU General Public License (GPL), the law of the Commonwealth of
   Virginia shall govern this License Agreement only as to issues arising under or
   with respect to Paragraphs 4, 5, and 7 of this License Agreement.  Nothing in
   this License Agreement shall be deemed to create any relationship of agency,
   partnership, or joint venture between CNRI and Licensee.  This License Agreement
   does not grant permission to use CNRI trademarks or trade name in a trademark
   sense to endorse or promote products or services of Licensee, or any third
   party.

8. By clicking on the "ACCEPT" button where indicated, or by copying, installing
   or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and
   conditions of this License Agreement.
```

C.2.4 LICENCE D'UTILISATION CWI POUR PYTHON 0.9.0 à 1.2

```
Copyright © 1991 - 1995, Stichting Mathematisch Centrum Amsterdam, The
Netherlands.  All rights reserved.

Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted, provided that
the above copyright notice appear in all copies and that both that copyright
notice and this permission notice appear in supporting documentation, and that
the name of Stichting Mathematisch Centrum or CWI not be used in advertising or
publicity pertaining to distribution of the software without specific, written
prior permission.

STICHTING MATHEMATISCH CENTRUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO
EVENT SHALL STICHTING MATHEMATISCH CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT
OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE,
DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS
SOFTWARE.
```

C.2.5 ZERO-CLAUSE BSD LICENSE FOR CODE IN THE PYTHON DOCUMENTATION

```
Permission to use, copy, modify, and/or distribute this software for any
purpose with or without fee is hereby granted.

THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH
REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY SPECIAL, DIRECT,
INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM
LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR
OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE OF THIS SOFTWARE.
```

C.3 Licences et remerciements pour les logiciels tiers

Cette section est une liste incomplète mais grandissante de licences et remerciements pour les logiciels tiers incorporés dans la distribution de Python.

C.3.1 Mersenne twister

The `_random` C extension underlying the *random* module includes code based on a download from http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html. The following are the verbatim comments from the original code :

```
A C-program for MT19937, with initialization improved 2002/1/26.
Coded by Takuji Nishimura and Makoto Matsumoto.

Before using, initialize the state by using init_genrand(seed)
or init_by_array(init_key, key_length).

Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura,
All rights reserved.

Redistribution and use in source and binary forms, with or without
```

(suite sur la page suivante)

(suite de la page précédente)

```
modification, are permitted provided that the following conditions
are met:

 1. Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

 2. Redistributions in binary form must reproduce the above copyright
    notice, this list of conditions and the following disclaimer in the
    documentation and/or other materials provided with the distribution.

 3. The names of its contributors may not be used to endorse or promote
    products derived from this software without specific prior written
    permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE COPYRIGHT OWNER OR
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.


Any feedback is very welcome.
http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html
email: m-mat @ math.sci.hiroshima-u.ac.jp (remove space)
```

C.3.2 Interfaces de connexion (*sockets*)

The `socket` module uses the functions, `getaddrinfo()`, and `getnameinfo()`, which are coded in separate source files from the WIDE Project, https://www.wide.ad.jp/.

```
Copyright (C) 1995, 1996, 1997, and 1998 WIDE Project.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.
3. Neither the name of the project nor the names of its contributors
   may be used to endorse or promote products derived from this software
   without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE PROJECT AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE PROJECT OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
```

(suite sur la page suivante)

(suite de la page précédente)

```
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.3 Interfaces de connexion asynchrones

The `test.support.asynchat` and `test.support.asyncore` modules contain the following notice :

```
Copyright 1996 by Sam Rushing

                         All Rights Reserved

Permission to use, copy, modify, and distribute this software and
its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of Sam
Rushing not be used in advertising or publicity pertaining to
distribution of the software without specific, written prior
permission.

SAM RUSHING DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN
NO EVENT SHALL SAM RUSHING BE LIABLE FOR ANY SPECIAL, INDIRECT OR
CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT,
NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN
CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
```

C.3.4 Gestion de témoin (*cookie*)

Le module `http.cookies` contient la note suivante :

```
Copyright 2000 by Timothy O'Malley <timo@alum.mit.edu>

               All Rights Reserved

Permission to use, copy, modify, and distribute this software
and its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Timothy O'Malley  not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.

Timothy O'Malley DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS, IN NO EVENT SHALL Timothy O'Malley BE LIABLE FOR
ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
```

(suite sur la page suivante)

(suite de la page précédente)

```
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE OF THIS SOFTWARE.
```

C.3.5 Traçage d'exécution

Le module `trace` contient la note suivante :

```
portions copyright 2001, Autonomous Zones Industries, Inc., all rights...
err...  reserved and offered to the public under the terms of the
Python 2.2 license.
Author: Zooko O'Whielacronx
http://zooko.com/
mailto:zooko@zooko.com

Copyright 2000, Mojam Media, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1999, Bioreason, Inc., all rights reserved.
Author: Andrew Dalke

Copyright 1995-1997, Automatrix, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1991-1995, Stichting Mathematisch Centrum, all rights reserved.


Permission to use, copy, modify, and distribute this Python software and
its associated documentation for any purpose without fee is hereby
granted, provided that the above copyright notice appears in all copies,
and that both that copyright notice and this permission notice appear in
supporting documentation, and that the name of neither Automatrix,
Bioreason or Mojam Media be used in advertising or publicity pertaining to
distribution of the software without specific, written prior permission.
```

C.3.6 Les fonctions UUencode et UUdecode

The `uu` codec contains the following notice :

```
Copyright 1994 by Lance Ellinghouse
Cathedral City, California Republic, United States of America.
                      All Rights Reserved
Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted,
provided that the above copyright notice appear in all copies and that
both that copyright notice and this permission notice appear in
supporting documentation, and that the name of Lance Ellinghouse
not be used in advertising or publicity pertaining to distribution
of the software without specific, written prior permission.
LANCE ELLINGHOUSE DISCLAIMS ALL WARRANTIES WITH REGARD TO
THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS, IN NO EVENT SHALL LANCE ELLINGHOUSE CENTRUM BE LIABLE
FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT
OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
```

(suite sur la page suivante)

(suite de la page précédente)

```
Modified by Jack Jansen, CWI, July 1995:
- Use binascii module to do the actual line-by-line conversion
  between ascii and binary. This results in a 1000-fold speedup. The C
  version is still 5 times faster, though.
- Arguments more compliant with Python standard
```

C.3.7 Appel de procédures distantes en XML (*RPC*, pour *Remote Procedure Call*)

Le module `xmlrpc.client` contient la note suivante :

```
    The XML-RPC client interface is

Copyright (c) 1999-2002 by Secret Labs AB
Copyright (c) 1999-2002 by Fredrik Lundh

By obtaining, using, and/or copying this software and/or its
associated documentation, you agree that you have read, understood,
and will comply with the following terms and conditions:

Permission to use, copy, modify, and distribute this software and
its associated documentation for any purpose and without fee is
hereby granted, provided that the above copyright notice appears in
all copies, and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Secret Labs AB or the author not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.

SECRET LABS AB AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD
TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANT-
ABILITY AND FITNESS.  IN NO EVENT SHALL SECRET LABS AB OR THE AUTHOR
BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY
DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE
OF THIS SOFTWARE.
```

C.3.8 test_epoll

The `test.test_epoll` module contains the following notice :

```
Copyright (c) 2001-2006 Twisted Matrix Laboratories.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
```

(suite sur la page suivante)

(suite de la page précédente)

```
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.9 Select kqueue

Le module `select` contient la note suivante pour l'interface *kqueue* :

```
Copyright (c) 2000 Doug White, 2006 James Knight, 2007 Christian Heimes
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.10 SipHash24

Le fichier `Python/pyhash.c` contient une implémentation par Marek Majkowski de l'algorithme *SipHash24* de Dan Bernstein. Il contient la note suivante :

```
<MIT License>
Copyright (c) 2013  Marek Majkowski <marek@popcount.org>

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.
</MIT License>

Original location:
   https://github.com/majek/csiphash/
```

(suite sur la page suivante)

(suite de la page précédente)

```
Solution inspired by code from:
   Samuel Neves (supercop/crypto_auth/siphash24/little)
   djb (supercop/crypto_auth/siphash24/little2)
   Jean-Philippe Aumasson (https://131002.net/siphash/siphash24.c)
```

C.3.11 *strtod* et *dtoa*

The file `Python/dtoa.c`, which supplies C functions dtoa and strtod for conversion of C doubles to and from strings, is derived from the file of the same name by David M. Gay, currently available from https://web.archive.org/web/20220517033456/http://www.netlib.org/fp/dtoa.c. The original file, as retrieved on March 16, 2009, contains the following copyright and licensing notice :

```
/****************************************************************
 *
 * The author of this software is David M. Gay.
 *
 * Copyright (c) 1991, 2000, 2001 by Lucent Technologies.
 *
 * Permission to use, copy, modify, and distribute this software for any
 * purpose without fee is hereby granted, provided that this entire notice
 * is included in all copies of any software which is or includes a copy
 * or modification of this software and in all copies of the supporting
 * documentation for such software.
 *
 * THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED
 * WARRANTY.  IN PARTICULAR, NEITHER THE AUTHOR NOR LUCENT MAKES ANY
 * REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY
 * OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.
 *
 ***************************************************************/
```

C.3.12 OpenSSL

The modules *hashlib*, *posix* and *ssl* use the OpenSSL library for added performance if made available by the operating system. Additionally, the Windows and macOS installers for Python may include a copy of the OpenSSL libraries, so we include a copy of the OpenSSL license here. For the OpenSSL 3.0 release, and later releases derived from that, the Apache License v2 applies :

```
                              Apache License
                        Version 2.0, January 2004
                     https://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

   "License" shall mean the terms and conditions for use, reproduction,
   and distribution as defined by Sections 1 through 9 of this document.

   "Licensor" shall mean the copyright owner or entity authorized by
   the copyright owner that is granting the License.

   "Legal Entity" shall mean the union of the acting entity and all
   other entities that control, are controlled by, or are under common
   control with that entity. For the purposes of this definition,
   "control" means (i) the power, direct or indirect, to cause the
```

(suite sur la page suivante)

(suite de la page précédente)

```
   direction or management of such entity, whether by contract or
   otherwise, or (ii) ownership of fifty percent (50%) or more of the
   outstanding shares, or (iii) beneficial ownership of such entity.

   "You" (or "Your") shall mean an individual or Legal Entity
   exercising permissions granted by this License.

   "Source" form shall mean the preferred form for making modifications,
   including but not limited to software source code, documentation
   source, and configuration files.

   "Object" form shall mean any form resulting from mechanical
   transformation or translation of a Source form, including but
   not limited to compiled object code, generated documentation,
   and conversions to other media types.

   "Work" shall mean the work of authorship, whether in Source or
   Object form, made available under the License, as indicated by a
   copyright notice that is included in or attached to the work
   (an example is provided in the Appendix below).

   "Derivative Works" shall mean any work, whether in Source or Object
   form, that is based on (or derived from) the Work and for which the
   editorial revisions, annotations, elaborations, or other modifications
   represent, as a whole, an original work of authorship. For the purposes
   of this License, Derivative Works shall not include works that remain
   separable from, or merely link (or bind by name) to the interfaces of,
   the Work and Derivative Works thereof.

   "Contribution" shall mean any work of authorship, including
   the original version of the Work and any modifications or additions
   to that Work or Derivative Works thereof, that is intentionally
   submitted to Licensor for inclusion in the Work by the copyright owner
   or by an individual or Legal Entity authorized to submit on behalf of
   the copyright owner. For the purposes of this definition, "submitted"
   means any form of electronic, verbal, or written communication sent
   to the Licensor or its representatives, including but not limited to
   communication on electronic mailing lists, source code control systems,
   and issue tracking systems that are managed by, or on behalf of, the
   Licensor for the purpose of discussing and improving the Work, but
   excluding communication that is conspicuously marked or otherwise
   designated in writing by the copyright owner as "Not a Contribution."

   "Contributor" shall mean Licensor and any individual or Legal Entity
   on behalf of whom a Contribution has been received by Licensor and
   subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   copyright license to reproduce, prepare Derivative Works of,
   publicly display, publicly perform, sublicense, and distribute the
   Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
```

(suite sur la page suivante)

(suite de la page précédente)

```
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   (except as stated in this section) patent license to make, have made,
   use, offer to sell, sell, import, and otherwise transfer the Work,
   where such license applies only to those patent claims licensable
   by such Contributor that are necessarily infringed by their
   Contribution(s) alone or by combination of their Contribution(s)
   with the Work to which such Contribution(s) was submitted. If You
   institute patent litigation against any entity (including a
   cross-claim or counterclaim in a lawsuit) alleging that the Work
   or a Contribution incorporated within the Work constitutes direct
   or contributory patent infringement, then any patent licenses
   granted to You under this License for that Work shall terminate
   as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the
   Work or Derivative Works thereof in any medium, with or without
   modifications, and in Source or Object form, provided that You
   meet the following conditions:

   (a) You must give any other recipients of the Work or
       Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices
       stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works
       that You distribute, all copyright, patent, trademark, and
       attribution notices from the Source form of the Work,
       excluding those notices that do not pertain to any part of
       the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its
       distribution, then any Derivative Works that You distribute must
       include a readable copy of the attribution notices contained
       within such NOTICE file, excluding those notices that do not
       pertain to any part of the Derivative Works, in at least one
       of the following places: within a NOTICE text file distributed
       as part of the Derivative Works; within the Source form or
       documentation, if provided along with the Derivative Works; or,
       within a display generated by the Derivative Works, if and
       wherever such third-party notices normally appear. The contents
       of the NOTICE file are for informational purposes only and
       do not modify the License. You may add Your own attribution
       notices within Derivative Works that You distribute, alongside
       or as an addendum to the NOTICE text from the Work, provided
       that such additional attribution notices cannot be construed
       as modifying the License.

   You may add Your own copyright statement to Your modifications and
   may provide additional or different license terms and conditions
   for use, reproduction, or distribution of Your modifications, or
   for any such Derivative Works as a whole, provided Your use,
   reproduction, and distribution of the Work otherwise complies with
   the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise,
```

(suite sur la page suivante)

(suite de la page précédente)

```
   any Contribution intentionally submitted for inclusion in the Work
   by You to the Licensor shall be under the terms and conditions of
   this License, without any additional terms or conditions.
   Notwithstanding the above, nothing herein shall supersede or modify
   the terms of any separate license agreement you may have executed
   with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade
   names, trademarks, service marks, or product names of the Licensor,
   except as required for reasonable and customary use in describing the
   origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or
   agreed to in writing, Licensor provides the Work (and each
   Contributor provides its Contributions) on an "AS IS" BASIS,
   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
   implied, including, without limitation, any warranties or conditions
   of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A
   PARTICULAR PURPOSE. You are solely responsible for determining the
   appropriateness of using or redistributing the Work and assume any
   risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory,
   whether in tort (including negligence), contract, or otherwise,
   unless required by applicable law (such as deliberate and grossly
   negligent acts) or agreed to in writing, shall any Contributor be
   liable to You for damages, including any direct, indirect, special,
   incidental, or consequential damages of any character arising as a
   result of this License or out of the use or inability to use the
   Work (including but not limited to damages for loss of goodwill,
   work stoppage, computer failure or malfunction, or any and all
   other commercial damages or losses), even if such Contributor
   has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing
   the Work or Derivative Works thereof, You may choose to offer,
   and charge a fee for, acceptance of support, warranty, indemnity,
   or other liability obligations and/or rights consistent with this
   License. However, in accepting such obligations, You may act only
   on Your own behalf and on Your sole responsibility, not on behalf
   of any other Contributor, and only if You agree to indemnify,
   defend, and hold each Contributor harmless for any liability
   incurred by, or claims asserted against, such Contributor by reason
   of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS
```

C.3.13 expat

The *pyexpat* extension is built using an included copy of the expat sources unless the build is configured `--with-system-expat` :

```
Copyright (c) 1998, 1999, 2000 Thai Open Source Software Center Ltd
                               and Clark Cooper

Permission is hereby granted, free of charge, to any person obtaining
```

(suite sur la page suivante)

(suite de la page précédente)

```
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.14 libffi

The `_ctypes` C extension underlying the *ctypes* module is built using an included copy of the libffi sources unless the build is configured `--with-system-libffi` :

```
Copyright (c) 1996-2008  Red Hat, Inc and others.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT.  IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.
```

C.3.15 zlib

Le module *zlib* est compilé en utilisant une copie du code source de *zlib* si la version de *zlib* trouvée sur le système est trop vieille pour être utilisée :

```
Copyright (C) 1995-2011 Jean-loup Gailly and Mark Adler

This software is provided 'as-is', without any express or implied
warranty.  In no event will the authors be held liable for any damages
arising from the use of this software.
```

(suite sur la page suivante)

(suite de la page précédente)

```
Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not
   claim that you wrote the original software. If you use this software
   in a product, an acknowledgment in the product documentation would be
   appreciated but is not required.

2. Altered source versions must be plainly marked as such, and must not be
   misrepresented as being the original software.

3. This notice may not be removed or altered from any source distribution.

Jean-loup Gailly        Mark Adler
jloup@gzip.org          madler@alumni.caltech.edu
```

C.3.16 cfuhash

L'implémentation des dictionnaires, utilisée par le module `tracemalloc` est basée sur le projet *cfuhash* :

```
Copyright (c) 2005 Don Owens
All rights reserved.

This code is released under the BSD license:

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

  * Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

  * Redistributions in binary form must reproduce the above
    copyright notice, this list of conditions and the following
    disclaimer in the documentation and/or other materials provided
    with the distribution.

  * Neither the name of the author nor the names of its
    contributors may be used to endorse or promote products derived
    from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
OF THE POSSIBILITY OF SUCH DAMAGE.
```

C.3.17 libmpdec

The `_decimal` C extension underlying the *decimal* module is built using an included copy of the libmpdec library unless the build is configured `--with-system-libmpdec` :

```
Copyright (c) 2008-2020 Stefan Krah. All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.18 Ensemble de tests C14N du W3C

Les tests de C14N version 2.0 du module *test* (`Lib/test/xmltestdata/c14n-20/`) proviennent du site du W3C à l'adresse https://www.w3.org/TR/xml-c14n2-testcases/ et sont distribués sous licence BSD modifiée :

```
Copyright (c) 2013 W3C(R) (MIT, ERCIM, Keio, Beihang),
All Rights Reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

* Redistributions of works must retain the original copyright notice,
  this list of conditions and the following disclaimer.
* Redistributions in binary form must reproduce the original copyright
  notice, this list of conditions and the following disclaimer in the
  documentation and/or other materials provided with the distribution.
* Neither the name of the W3C nor the names of its contributors may be
  used to endorse or promote products derived from this work without
  specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
```

(suite sur la page suivante)

(suite de la page précédente)

```
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

C.3.19 mimalloc

MIT License :

```
Copyright (c) 2018-2021 Microsoft Corporation, Daan Leijen

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

C.3.20 *asyncio*

Parts of the `asyncio` module are incorporated from uvloop 0.16, which is distributed under the MIT license :

```
Copyright (c) 2015-2021 MagicStack Inc.  http://magic.io

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.21 Global Unbounded Sequences (GUS)

The file `Python/qsbr.c` is adapted from FreeBSD's "Global Unbounded Sequences" safe memory reclamation scheme in subr_smr.c. The file is distributed under the 2-Clause BSD License :


```
Copyright (c) 2019,2020 Jeffrey Roberson <jeff@FreeBSD.org>

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice unmodified, this list of conditions, and the following
   disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR "AS IS" AND ANY EXPRESS OR
IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.
IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT,
INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF
THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

ANNEXE D

Copyright

Python et cette documentation sont :

Copyright © 2001 Python Software Foundation. All rights reserved.

Copyright © 2000 *BeOpen.com*. Tous droits réservés.

Copyright © 1995-2000 *Corporation for National Research Initiatives*. Tous droits réservés.

Copyright © 1991-1995 *Stichting Mathematisch Centrum*. Tous droits réservés.

Voir *Histoire et licence* pour des informations complètes concernant la licence et les permissions.

Bibliographie

[Frie09] *Friedl, Jeffrey. Mastering Regular Expressions. 3rd ed., O'Reilly Media, 2009*. La troisième édition de ce livre ne couvre plus du tout Python, mais la première version explique en détails comment écrire de bonnes expressions rationnelles.

[C99] ISO/IEC 9899 :1999. "Programming languages -- C." A public draft of this standard is available at https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf.

Index des modules Python

_

__future__, 2020
__main__, 1966
_thread, 1056
_tkinter, 1622

a

abc, 2004
aifc, 2221
annotationlib, 2045
argparse, 849
array, 289
ast, 2101
asynchat, 2221
asyncio, 1061
asyncore, 2221
atexit, 2009
audioop, 2222

b

base64, 1339
bdb, 1869
binascii, 1343
bisect, 286
builtins, 1965
bz2, 574

c

calendar, 250
cgi, 2222
cgitb, 2222
chunk, 2222
cmath, 347
cmd, 1608
code, 2053
codecs, 189
codeop, 2055
collections, 258
collections.abc, 276
colorsys, 1552
compileall, 2156
concurrent.futures, 1020
configparser, 620
contextlib, 1989
contextvars, 1052
copy, 306
copyreg, 525
cProfile, 1888
crypt, 2222
csv, 613
ctypes, 812
curses (*Unix*), 923
curses.ascii, 949
curses.panel, 953
curses.textpad, 948

d

dataclasses, 1978
datetime, 207
dbm, 531
dbm.dumb, 535
dbm.gnu (*Unix*), 533
dbm.ndbm (*Unix*), 534
dbm.sqlite3 (*All*), 532
decimal, 351
difflib, 155
dis, 2160
distutils, 2222
doctest, 1728

e

email, 1249
email.charset, 1301
email.contentmanager, 1279
email.encoders, 1303
email.errors, 1272
email.generator, 1262
email.header, 1298
email.headerregistry, 1273
email.iterators, 1306
email.message, 1250
email.mime, 1296
email.mime.application, 1297
email.mime.audio, 1297
email.mime.base, 1296
email.mime.image, 1297
email.mime.message, 1298

email.mime.multipart, 1296
email.mime.nonmultipart, 1296
email.mime.text, 1298
email.parser, 1258
email.policy, 1264
email.utils, 1304
encodings.idna, 204
encodings.mbcs, 205
encodings.utf_8_sig, 205
ensurepip, 1913
enum, 316
errno, 804

f

faulthandler, 1874
fcntl (*Unix*), 2204
filecmp, 485
fileinput, 921
fnmatch, 495
fractions, 380
ftplib, 1465
functools, 426

g

gc, 2021
getopt, 2217
getpass, 920
gettext, 1553
glob, 492
graphlib, 331
grp (*Unix*), 2199
gzip, 571

h

hashlib, 645
heapq, 282
hmac, 657
html, 1347
html.entities, 1352
html.parser, 1347
http, 1454
http.client, 1457
http.cookiejar, 1512
http.cookies, 1509
http.server, 1502

i

idlelib, 1670
imaplib, 1474
imghdr, 2223
imp, 2223
importlib, 2066
importlib.abc, 2068
importlib.machinery, 2074
importlib.metadata, 2090
importlib.resources, 2085
importlib.resources.abc, 2088
importlib.util, 2079
inspect, 2025
io, 730
ipaddress, 1534
itertools, 409

j

json, 1307
json.tool, 1316

k

keyword, 2147

l

linecache, 496
locale, 1561
logging, 756
logging.config, 774
logging.handlers, 786
lzma, 579

m

mailbox, 1317
mailcap, 2223
marshal, 529
math, 338
mimetypes, 1336
mmap, 1243
modulefinder, 2062
msilib, 2223
msvcrt (*Windows*), 2183
multiprocessing, 969
multiprocessing.connection, 1001
multiprocessing.dummy, 1004
multiprocessing.managers, 991
multiprocessing.pool, 998
multiprocessing.shared_memory, 1014
multiprocessing.sharedctypes, 989

n

netrc, 641
nis, 2223
nntplib, 2223
numbers, 335

o

operator, 437
optparse, 894
os, 663
os.path, 472
ossaudiodev, 2224

p

pathlib, 445
pathlib.types, 472
pdb, 1876
pickle, 509
pickletools, 2181
pipes, 2224

pkgutil, 2059
platform, 800
plistlib, 642
poplib, 1471
posix (*Unix*), 2197
pprint, 307
profile, 1888
pstats, 1889
pty (*Unix*), 2202
pwd (*Unix*), 2198
py_compile, 2154
pyclbr, 2152
pydoc, 1724

q

queue, 1049
quopri, 1345

r

random, 383
re, 132
readline (*Unix*), 175
reprlib, 313
resource (*Unix*), 2206
rlcompleter, 180
runpy, 2063

s

sched, 1047
secrets, 659
select, 1224
selectors, 1231
shelve, 526
shlex, 1613
shutil, 497
signal, 1234
site, 2049
sitecustomize, 2050
smtpd, 2224
smtplib, 1482
sndhdr, 2224
socket, 1162
socketserver, 1493
spwd, 2224
sqlite3, 536
ssl, 1189
stat, 479
statistics, 393
string, 121
stringprep, 174
struct, 181
subprocess, 1028
sunau, 2224
symtable, 2139
sys, 1929
sys.monitoring, 1955
sysconfig, 1959
syslog (*Unix*), 2210

t

tabnanny, 2151
tarfile, 595
telnetlib, 2225
tempfile, 487
termios (*Unix*), 2200
test, 1844
test.regrtest, 1846
test.support, 1847
test.support.bytecode_helper, 1858
test.support.import_helper, 1861
test.support.os_helper, 1859
test.support.script_helper, 1857
test.support.socket_helper, 1856
test.support.threading_helper, 1858
test.support.warnings_helper, 1862
textwrap, 168
threading, 955
time, 743
timeit, 1893
tkinter, 1619
tkinter.colorchooser (*Tk*), 1632
tkinter.commondialog (*Tk*), 1637
tkinter.dnd (*Tk*), 1639
tkinter.filedialog (*Tk*), 1634
tkinter.font (*Tk*), 1633
tkinter.messagebox (*Tk*), 1637
tkinter.scrolledtext (*Tk*), 1639
tkinter.simpledialog (*Tk*), 1634
tkinter.ttk, 1640
token, 2143
tokenize, 2147
tomllib, 639
trace, 1898
traceback, 2010
tracemalloc, 1901
tty (*Unix*), 2201
turtle, 1571
turtledemo, 1606
types, 299
typing, 1671

u

unicodedata, 172
unittest, 1751
unittest.mock, 1783
urllib, 1426
urllib.error, 1452
urllib.parse, 1443
urllib.request, 1426
urllib.response, 1442
urllib.robotparser, 1452
usercustomize, 2050
uu, 2225
uuid, 1488

v

venv, 1915

W

warnings, 1971
wave, 1549
weakref, 292
webbrowser, 1413
winreg (*Windows*), 2186
winsound (*Windows*), 2194
wsgiref, 1416
wsgiref.handlers, 1421
wsgiref.headers, 1418
wsgiref.simple_server, 1419
wsgiref.types, 1424
wsgiref.util, 1416
wsgiref.validate, 1420

X

xdrlib, 2225
xml, 1353
xml.dom, 1374
xml.dom.minidom, 1384
xml.dom.pulldom, 1389
xml.etree.ElementInclude, 1366
xml.etree.ElementTree, 1354
xml.parsers.expat, 1403
xml.parsers.expat.errors, 1410
xml.parsers.expat.model, 1409
xml.sax, 1391
xml.sax.handler, 1392
xml.sax.saxutils, 1398
xml.sax.xmlreader, 1399
xmlrpc, 1521
xmlrpc.client, 1521
xmlrpc.server, 1528

Z

zipapp, 1924
zipfile, 584
zipimport, 2057
zlib, 567
zoneinfo, 245

Index

Non alphabétique

??
    in regular expressions, 134
..
    in pathnames, 728
..., **2229**
    ellipsis literal, 36, 103
    in doctests, 1736
    interpreter prompt, 1733, 1948
    placeholder, 172, 308, 313
. (*dot*)
    in glob-style wildcards, 492
    in pathnames, 728
    in printf-style formatting, 63, 79
    in regular expressions, 133
    in string formatting, 123
! (*exclamation*)
    in a command interpreter, 1608
    in curses module, 953
    in glob-style wildcards, 492, 495
    in string formatting, 123
    in struct format strings, 182
- (*minus*)
    binary operator, 39
    in doctests, 1737
    in glob-style wildcards, 492, 495
    in printf-style formatting, 63, 79
    in regular expressions, 135
    in string formatting, 125
    unary operator, 39
! (*pdb command*), 1884
? (*question mark*)
    in a command interpreter, 1608
    in argparse module, 861
    in AST grammar, 2104
    in glob-style wildcards, 492, 495
    in regular expressions, 134
    in SQL statements, 552
    in struct format strings, 184, 185
    replacement character, 192
(*hash*)
    comment, 2049
    in doctests, 1737
    in printf-style formatting, 63, 79
    in regular expressions, 141
    in string formatting, 126
$ (*dollar*)
    environment variables expansion, 474
    in regular expressions, 133
    in template strings, 130
    interpolation in configuration files, 626
% (*percent*)
    datetime format, 240, 748, 750
    environment variables expansion (*Windows*), 474, 2188
    interpolation in configuration files, 625
    operator, 39
    printf-style formatting, 63, 79
& (*ampersand*)
    operator, 40
(?
    in regular expressions, 135
(?!
    in regular expressions, 137
(?#
    in regular expressions, 137
(?(
    in regular expressions, 138
() (*parentheses*)
    in printf-style formatting, 63, 79
    in regular expressions, 135
(?:
    in regular expressions, 136
(?<!
    in regular expressions, 138
(?<=
    in regular expressions, 137
(?=
    in regular expressions, 137
(?P<
    in regular expressions, 136
(?P=
    in regular expressions, 137
*?
    in regular expressions, 134

* (*asterisk*)
    in argparse module, 861
    in AST grammar, 2104
    in glob-style wildcards, 492, 495
    in printf-style formatting, 63, 79
    in regular expressions, 133
    operator, 39
**
    in glob-style wildcards, 493
    operator, 39
*+
    in regular expressions, 134
Clear Breakpoint, 1663
Copy, 1663
Cut, 1663
IDLE, 1659
Paste, 1663
Set Breakpoint, 1663
+?
    in regular expressions, 134
?+
    in regular expressions, 134
+ (*plus*)
    binary operator, 39
    in argparse module, 862
    in doctests, 1737
    in printf-style formatting, 63, 79
    in regular expressions, 133
    in string formatting, 125
    unary operator, 39
++
    in regular expressions, 134
, (*comma*)
    in string formatting, 126
-
    option de ligne de commande
        python--m-py_compile, 2155
/ (*slash*)
    in pathnames, 728
    operator, 39
//
    operator, 39
2-digit years, 744
: (*colon*)
    in SQL statements, 552
    in string formatting, 123
    path separator (*POSIX*), 728
; (*semicolon*), 728
< (*less*)
    in string formatting, 125
    in struct format strings, 182
    operator, 38
<<
    operator, 40
<=
    operator, 38
<BLANKLINE>, 1735
<file>
    option de ligne de commande
        python--m-py_compile, 2155
!=
    operator, 38
= (*equals*)
    in string formatting, 125
    in struct format strings, 182
==
    operator, 38
> (*greater*)
    in string formatting, 125
    in struct format strings, 182
    operator, 38
>=
    operator, 38
>>
    operator, 40
>>>, **2229**
    interpreter prompt, 1733, 1948
@ (*at*)
    in struct format strings, 182
[] (*square brackets*)
    in glob-style wildcards, 492, 495
    in regular expressions, 135
    in string formatting, 123
\ (*backslash*)
    escape sequence, 192
    in pathnames (*Windows*), 728
    in regular expressions, 134, 135, 138
\\
    in regular expressions, 139
\A
    in regular expressions, 138
\a
    in regular expressions, 139
\B
    in regular expressions, 138
\b
    in regular expressions, 138, 139
\D
    in regular expressions, 139
\d
    in regular expressions, 138
\f
    in regular expressions, 139
\g
    in regular expressions, 144
\N
    escape sequence, 192
    in regular expressions, 139
\n
    in regular expressions, 139
\r
    in regular expressions, 139
\S
    in regular expressions, 139
\s
    in regular expressions, 139

\t
    in regular expressions, 139
\U
    escape sequence, 192
    in regular expressions, 139
\u
    escape sequence, 192
    in regular expressions, 139
\v
    in regular expressions, 139
\W
    in regular expressions, 139
\w
    in regular expressions, 139
\x
    escape sequence, 192
    in regular expressions, 139
\Z
    in regular expressions, 139
^ (*caret*)
    in curses module, 953
    in regular expressions, 133, 135
    in string formatting, 125
    marker, 1735, 2011
    operator, 40
_ (*underscore*)
    gettext, 1554
    in string formatting, 126
__abs__() (*dans le module operator*), 438
__add__() (*dans le module operator*), 438
__and__() (*dans le module operator*), 438
__and__() (*méthode enum.Flag*), 324
__args__ (*attribut genericalias*), 99
__bound__ (*attribut typing.TypeVar*), 1697
__breakpointhook__ (*dans le module sys*), 1934
__bytes__() (*méthode email.message.EmailMessage*), 1251
__bytes__() (*méthode email.message.Message*), 1289
__call__() (*dans le module operator*), 440
__call__() (*méthode argparse.Action*), 867
__call__() (*méthode email.headerregistry.HeaderRegistry*), 1277
__call__() (*méthode enum.EnumType*), 318
__call__() (*méthode weakref.finalize*), 295
__callback__ (*attribut weakref.ref*), 293
__cause__ (*attribut BaseException*), 107
__cause__ (*attribut traceback.TracebackException*), 2013
__cause__ (*exception attribute*), 107
__ceil__() (*méthode fractions.Fraction*), 382
__class__ (*attribut unittest.mock.Mock*), 1794
__code__ (*function object attribute*), 103
__concat__() (*dans le module operator*), 439
__constraints__ (*attribut typing.TypeVar*), 1697
__contains__() (*dans le module operator*), 439
__contains__() (*méthode email.message.EmailMessage*), 1252
__contains__() (*méthode email.message.Message*), 1291
__contains__() (*méthode enum.EnumType*), 318
__contains__() (*méthode enum.Flag*), 323
__contains__() (*méthode mailbox.Mailbox*), 1319
__context__ (*attribut BaseException*), 107
__context__ (*attribut traceback.TracebackException*), 2013
__context__ (*exception attribute*), 107
__contravariant__ (*attribut typing.TypeVar*), 1697
__copy__() (*copy protocol*), 306
__covariant__ (*attribut typing.TypeVar*), 1697
__debug__ (*variable de base*), 36
__deepcopy__() (*copy protocol*), 306
__default__ (*attribut typing.ParamSpec*), 1700
__default__ (*attribut typing.TypeVar*), 1697
__default__ (*attribut typing.TypeVarTuple*), 1699
__del__() (*méthode io.IOBase*), 735
__delitem__() (*dans le module operator*), 440
__delitem__() (*méthode email.message.EmailMessage*), 1252
__delitem__() (*méthode email.message.Message*), 1291
__delitem__() (*méthode mailbox.Mailbox*), 1318
__delitem__() (*méthode mailbox.MH*), 1324
__dir__() (*méthode enum.Enum*), 320
__dir__() (*méthode enum.EnumType*), 318
__dir__() (*méthode unittest.mock.Mock*), 1790
__displayhook__ (*dans le module sys*), 1934
__doc__ (*attribut definition*), 104
__enter__() (*méthode contextmanager*), 95
__enter__() (*méthode winreg.PyHKEY*), 2194
__eq__() (*dans le module operator*), 437
__eq__() (*instance method*), 38
__eq__() (*méthode email.charset.Charset*), 1302
__eq__() (*méthode email.header.Header*), 1300
__eq__() (*méthode memoryview*), 82
__excepthook__ (*dans le module sys*), 1934
__excepthook__ (*dans le module threading*), 956
__exit__() (*méthode contextmanager*), 96
__exit__() (*méthode winreg.PyHKEY*), 2194
__floor__() (*méthode fractions.Fraction*), 382
__floordiv__() (*dans le module operator*), 438
__format__, 17
__format__() (*méthode datetime.date*), 217
__format__() (*méthode datetime.datetime*), 227
__format__() (*méthode datetime.time*), 232
__format__() (*méthode enum.Enum*), 322
__format__() (*méthode fractions.Fraction*), 382
__format__() (*méthode ipaddress.IPv4Address*), 1537
__format__() (*méthode ipaddress.IPv6Address*), 1539
__forward_arg__ (*attribut annotationlib.ForwardRef*), 2046
__fspath__() (*méthode os.PathLike*), 666
__future__, **2235**
    module, 2020
__ge__() (*dans le module operator*), 437
__ge__() (*instance method*), 38

__getitem__() (*dans le module operator*), 440
__getitem__() (*méthode email.headerregistry.HeaderRegistry*), 1277
__getitem__() (*méthode email.message.EmailMessage*), 1252
__getitem__() (*méthode email.message.Message*), 1291
__getitem__() (*méthode enum.EnumType*), 318
__getitem__() (*méthode mailbox.Mailbox*), 1319
__getitem__() (*méthode re.Match*), 148
__getnewargs__() (*méthode object*), 516
__getnewargs_ex__() (*méthode object*), 515
__getstate__() (*copy protocol*), 520
__getstate__() (*méthode object*), 516
__gt__() (*dans le module operator*), 437
__gt__() (*instance method*), 38
__iadd__() (*dans le module operator*), 443
__iand__() (*dans le module operator*), 443
__iconcat__() (*dans le module operator*), 443
__ifloordiv__() (*dans le module operator*), 443
__ilshift__() (*dans le module operator*), 443
__imatmul__() (*dans le module operator*), 443
__imod__() (*dans le module operator*), 443
__import__() built-in function, 33
__import__() (*dans le module importlib*), 2067
__imul__() (*dans le module operator*), 443
__index__() (*dans le module operator*), 438
__infer_variance__ (*attribut typing.TypeVar*), 1697
__init__() (*méthode asyncio.Future*), 1151
__init__() (*méthode asyncio.Task*), 1151
__init__() (*méthode difflib.HtmlDiff*), 156
__init__() (*méthode enum.Enum*), 321
__init__() (*méthode logging.Handler*), 762
__init_subclass__() (*méthode enum.Enum*), 321
__interactivehook__ (*dans le module sys*), 1944
__inv__() (*dans le module operator*), 438
__invert__() (*dans le module operator*), 438
__ior__() (*dans le module operator*), 443
__ipow__() (*dans le module operator*), 443
__irshift__() (*dans le module operator*), 444
__isub__() (*dans le module operator*), 444
__iter__() (*méthode container*), 46
__iter__() (*méthode enum.EnumType*), 319
__iter__() (*méthode iterator*), 46
__iter__() (*méthode mailbox.Mailbox*), 1318
__iter__() (*méthode unittest.TestSuite*), 1773
__itruediv__() (*dans le module operator*), 444
__ixor__() (*dans le module operator*), 444
__le__() (*dans le module operator*), 437
__le__() (*instance method*), 38
__len__() (*méthode email.message.EmailMessage*), 1252
__len__() (*méthode email.message.Message*), 1291
__len__() (*méthode enum.EnumType*), 319
__len__() (*méthode mailbox.Mailbox*), 1319
__lshift__() (*dans le module operator*), 439
__lt__() (*dans le module operator*), 437
__lt__() (*instance method*), 38
__main__ module, 1966, 2064
__matmul__() (*dans le module operator*), 439
__members__ (*attribut enum.EnumType*), 319
__missing__(), 91
__missing__() (*méthode collections.defaultdict*), 268
__mod__() (*dans le module operator*), 439
__module__ (*attribut definition*), 104
__module__ (*attribut typing.NewType*), 1704
__module__ (*attribut typing.TypeAliasType*), 1702
__mul__() (*dans le module operator*), 439
__mutable_keys__ (*attribut typing.TypedDict*), 1709
__name__ (*attribut definition*), 103
__name__ (*attribut property*), 28
__name__ (*attribut typing.NewType*), 1704
__name__ (*attribut typing.ParamSpec*), 1700
__name__ (*attribut typing.TypeAliasType*), 1702
__name__ (*attribut typing.TypeVar*), 1696
__name__ (*attribut typing.TypeVarTuple*), 1699
__ne__() (*dans le module operator*), 437
__ne__() (*instance method*), 38
__ne__() (*méthode email.charset.Charset*), 1302
__ne__() (*méthode email.header.Header*), 1300
__neg__() (*dans le module operator*), 439
__new__() (*méthode enum.Enum*), 321
__next__() (*méthode csv.csvreader*), 618
__next__() (*méthode iterator*), 46
__not__() (*dans le module operator*), 438
__notes__ (*attribut BaseException*), 109
__notes__ (*attribut traceback.TracebackException*), 2014
__optional_keys__ (*attribut typing.TypedDict*), 1708
__or__() (*dans le module operator*), 439
__or__() (*méthode enum.Flag*), 324
__origin__ (*attribut genericalias*), 99
__parameters__ (*attribut genericalias*), 99
__pos__() (*dans le module operator*), 439
__post_init__() (*dans le module dataclasses*), 1985
__pow__() (*dans le module operator*), 439
__qualname__ (*attribut definition*), 104
__readonly_keys__ (*attribut typing.TypedDict*), 1709
__reduce__() (*méthode object*), 516
__reduce_ex__() (*méthode object*), 517
__replace__() (*replace protocol*), 307
__repr__() (*méthode enum.Enum*), 321
__repr__() (*méthode multiprocessing.managers.BaseProxy*), 997
__repr__() (*méthode netrc.netrc*), 642
__required_keys__ (*attribut typing.TypedDict*), 1708
__reversed__() (*méthode enum.EnumType*), 319
__round__() (*méthode fractions.Fraction*), 382
__rshift__() (*dans le module operator*), 439
__setitem__() (*dans le module operator*), 440
__setitem__() (*méthode email.message.EmailMessage*), 1252
__setitem__() (*méthode email.message.Message*), 1291

__setitem__() (*méthode mailbox.Mailbox*), 1318
__setitem__() (*méthode mailbox.Maildir*), 1322
__setstate__() (*copy protocol*), 520
__setstate__() (*méthode object*), 516
__slots__, **2242**
__stderr__ (*dans le module sys*), 1952
__stdin__ (*dans le module sys*), 1952
__stdout__ (*dans le module sys*), 1952
__str__() (*méthode datetime.date*), 216
__str__() (*méthode datetime.datetime*), 226
__str__() (*méthode datetime.time*), 232
__str__() (*méthode email.charset.Charset*), 1302
__str__() (*méthode email.header.Header*), 1300
__str__() (*méthode email.headerregistry.Address*), 1278
__str__() (*méthode email.headerregistry.Group*), 1278
__str__() (*méthode email.message.EmailMessage*), 1251
__str__() (*méthode email.message.Message*), 1289
__str__() (*méthode enum.Enum*), 322
__str__() (*méthode multiprocessing.managers.BaseProxy*), 997
__sub__() (*dans le module operator*), 439
__subclasshook__() (*méthode abc.ABCMeta*), 2005
__supertype__ (*attribut typing.NewType*), 1704
__suppress_context__ (*attribut BaseException*), 107
__suppress_context__ (*attribut traceback.TracebackException*), 2013
__suppress_context__ (*exception attribute*), 107
__total__ (*attribut typing.TypedDict*), 1708
__traceback__ (*attribut BaseException*), 108
__truediv__() (*dans le module operator*), 439
__truediv__() (*méthode importlib.abc.Traversable*), 2073
__truediv__() (*méthode importlib.resources.abc.Traversable*), 2089
__type_params__ (*attribut definition*), 104
__type_params__ (*attribut typing.TypeAliasType*), 1702
__unpacked__ (*attribut genericalias*), 99
__unraisablehook__ (*dans le module sys*), 1934
__value__ (*attribut typing.TypeAliasType*), 1702
__version__ (*dans le module curses*), 937
__xor__() (*dans le module operator*), 439
__xor__() (*méthode enum.Flag*), 324
_add_alias_() (*méthode enum.EnumType*), 319
_add_value_alias_() (*méthode enum.EnumType*), 319
align (*attribut ctypes.Structure*), 847
anonymous (*attribut ctypes.Structure*), 847
_asdict() (*méthode collections.somenamedtuple*), 271
_b_base_ (*attribut ctypes._CData*), 842
_b_needsfree_ (*attribut ctypes._CData*), 843
_callmethod() (*méthode multiprocessing.managers.BaseProxy*), 997
_CData (*classe dans ctypes*), 842
_CFuncPtr (*classe dans ctypes*), 835
_clear_internal_caches() (*dans le module sys*), 1931
_clear_type_cache() (*dans le module sys*), 1931
_current_exceptions() (*dans le module sys*), 1931
_current_frames() (*dans le module sys*), 1931
_debugmallocstats() (*dans le module sys*), 1932
_emscripten_info (*dans le module sys*), 1933
_enablelegacywindowsfsencoding() (*dans le module sys*), 1951
_enter_task() (*dans le module asyncio*), 1152
_exit() (*dans le module os*), 713
_Feature (*classe dans __future__*), 2021
_field_defaults (*attribut collections.somenamedtuple*), 271
_field_types (*attribut ast.AST*), 2105
_fields (*attribut ast.AST*), 2104
_fields (*attribut collections.somenamedtuple*), 271
fields (*attribut ctypes.Structure*), 846
_flush() (*méthode wsgiref.handlers.BaseHandler*), 1422
_for_archive() (*méthode zipfile.ZipInfo*), 585
_generate_next_value_() (*méthode enum.Enum*), 320
_get_child_mock() (*méthode unittest.mock.Mock*), 1790
_get_preferred_schemes() (*dans le module sysconfig*), 1963
_getframe() (*dans le module sys*), 1941
_getframemodulename() (*dans le module sys*), 1941
_getvalue() (*méthode multiprocessing.managers.BaseProxy*), 997
_handle (*attribut ctypes.PyDLL*), 834
ignore (*attribut enum.Enum*), 320
_incompatible_extension_module_restrictions() (*dans le module importlib.util*), 2081
_is_gil_enabled() (*dans le module sys*), 1944
_is_immortal() (*dans le module sys*), 1945
_is_interned() (*dans le module sys*), 1945
layout (*attribut ctypes.Structure*), 847
_leave_task() (*dans le module asyncio*), 1152
length (*attribut ctypes.Array*), 848
_locale
module, 1561
_log (*attribut logging.LoggerAdapter*), 769
_make() (*méthode de la classe collections.somenamedtuple*), 271
_makeResult() (*méthode unittest.TextTestRunner*), 1779
missing() (*méthode enum.Enum*), 321
_name (*attribut ctypes.PyDLL*), 834
name (*attribut enum.Enum*), 320
_numeric_repr_() (*méthode enum.Flag*), 325
_objects (*attribut ctypes._CData*), 843
order (*attribut enum.Enum*), 320
pack (*attribut ctypes.Structure*), 846
_parse() (*méthode gettext.NullTranslations*), 1555
_Pointer (*classe dans ctypes*), 848
_register_task() (*dans le module asyncio*), 1152

_replace() (*méthode collections.somenamedtuple*), 271
_setroot() (*méthode xml.etree.ElementTree.ElementTree*), 1369
_SimpleCData (*classe dans ctypes*), 843
_structure() (*dans le module email.iterators*), 1307
_thread
    module, 1056
_tkinter
    module, 1622
type (*attribut ctypes._Pointer*), 848
type (*attribut ctypes.Array*), 848
_unregister_task() (*dans le module asyncio*), 1152
value (*attribut enum.Enum*), 320
_write() (*méthode wsgiref.handlers.BaseHandler*), 1422
_xoptions (*dans le module sys*), 1954
{} (*curly brackets*)
    in regular expressions, 134
    in string formatting, 123
| (*vertical bar*)
    in regular expressions, 135
    operator, 40
~ (*tilde*)
    home directory expansion, 474
    operator, 40

A

-a
    option de ligne de commande ast, 2139
    option de ligne de commande pickletools, 2182
A (*dans le module re*), 140
a2b_base64() (*dans le module binascii*), 1343
a2b_hex() (*dans le module binascii*), 1344
a2b_qp() (*dans le module binascii*), 1343
a2b_uu() (*dans le module binascii*), 1343
a85decode() (*dans le module base64*), 1341
a85encode() (*dans le module base64*), 1341
A_ALTCHARSET (*dans le module curses*), 938
A_ATTRIBUTES (*dans le module curses*), 939
A_BLINK (*dans le module curses*), 938
A_BOLD (*dans le module curses*), 938
A_CHARTEXT (*dans le module curses*), 939
A_COLOR (*dans le module curses*), 939
A_DIM (*dans le module curses*), 938
A_HORIZONTAL (*dans le module curses*), 938
A_INVIS (*dans le module curses*), 938
A_ITALIC (*dans le module curses*), 938
A_LEFT (*dans le module curses*), 938
A_LOW (*dans le module curses*), 938
A_NORMAL (*dans le module curses*), 938
A_PROTECT (*dans le module curses*), 938
A_REVERSE (*dans le module curses*), 938
A_RIGHT (*dans le module curses*), 938
A_STANDOUT (*dans le module curses*), 938
A_TOP (*dans le module curses*), 938
A_UNDERLINE (*dans le module curses*), 938
A_VERTICAL (*dans le module curses*), 938
abc
    module, 2004
ABC (*classe dans abc*), 2004
ABCMeta (*classe dans abc*), 2004
ABDAY_1 (*dans le module locale*), 1563
ABDAY_2 (*dans le module locale*), 1563
ABDAY_3 (*dans le module locale*), 1563
ABDAY_4 (*dans le module locale*), 1563
ABDAY_5 (*dans le module locale*), 1563
ABDAY_6 (*dans le module locale*), 1563
ABDAY_7 (*dans le module locale*), 1563
abiflags (*dans le module sys*), 1929
ABMON_1 (*dans le module locale*), 1564
ABMON_2 (*dans le module locale*), 1564
ABMON_3 (*dans le module locale*), 1564
ABMON_4 (*dans le module locale*), 1564
ABMON_5 (*dans le module locale*), 1564
ABMON_6 (*dans le module locale*), 1564
ABMON_7 (*dans le module locale*), 1564
ABMON_8 (*dans le module locale*), 1564
ABMON_9 (*dans le module locale*), 1564
ABMON_10 (*dans le module locale*), 1564
ABMON_11 (*dans le module locale*), 1564
ABMON_12 (*dans le module locale*), 1564
ABORT (*dans le module tkinter.messagebox*), 1638
abort() (*dans le module os*), 712
abort() (*méthode asyncio.Barrier*), 1097
abort() (*méthode asyncio.DatagramTransport*), 1138
abort() (*méthode asyncio.WriteTransport*), 1137
abort() (*méthode ftplib.FTP*), 1467
abort() (*méthode threading.Barrier*), 968
abort_clients() (*méthode asyncio.Server*), 1126
ABORTRETRYIGNORE (*dans le module tkinter.messagebox*), 1638
above() (*méthode curses.panel.Panel*), 953
ABOVE_NORMAL_PRIORITY_CLASS (*dans le module subprocess*), 1041
abs()
    built-in function, 8
abs() (*dans le module operator*), 438
abs() (*méthode decimal.Context*), 366
absolute() (*méthode pathlib.Path*), 459
AbsoluteLinkError, 598
AbsolutePathError, 598
abspath() (*dans le module os.path*), 473
AbstractAsyncContextManager (*classe dans contextlib*), 1989
AbstractBasicAuthHandler (*classe dans urllib.request*), 1430
abstractclassmethod() (*dans le module abc*), 2007
AbstractContextManager (*classe dans contextlib*), 1989
AbstractDigestAuthHandler (*classe dans urllib.request*), 1430
AbstractEventLoop (*classe dans asyncio*), 1128
AbstractEventLoopPolicy (*classe dans asyncio*), 1148

abstractmethod() (*dans le module abc*), 2006
abstractproperty() (*dans le module abc*), 2007
AbstractSet (*classe dans typing*), 1721
abstractstaticmethod() (*dans le module abc*), 2007
accept() (*méthode multiprocessing.connection.Listener*), 1001
accept() (*méthode socket.socket*), 1178
access() (*dans le module os*), 686
accumulate() (*dans le module itertools*), 411
ACK (*dans le module curses.ascii*), 950
aclose() (*méthode contextlib.AsyncExitStack*), 1998
aclosing() (*dans le module contextlib*), 1992
acos() (*dans le module cmath*), 349
acos() (*dans le module math*), 345
acosh() (*dans le module cmath*), 349
acosh() (*dans le module math*), 345
acquire() (*méthode _thread.lock*), 1058
acquire() (*méthode asyncio.Condition*), 1094
acquire() (*méthode asyncio.Lock*), 1092
acquire() (*méthode asyncio.Semaphore*), 1095
acquire() (*méthode logging.Handler*), 762
acquire() (*méthode multiprocessing.Lock*), 987
acquire() (*méthode multiprocessing.RLock*), 987
acquire() (*méthode threading.Condition*), 964
acquire() (*méthode threading.Lock*), 961
acquire() (*méthode threading.RLock*), 962
acquire() (*méthode threading.Semaphore*), 965
ACS_BBSS (*dans le module curses*), 945
ACS_BLOCK (*dans le module curses*), 945
ACS_BOARD (*dans le module curses*), 945
ACS_BSBS (*dans le module curses*), 945
ACS_BSSB (*dans le module curses*), 945
ACS_BSSS (*dans le module curses*), 945
ACS_BTEE (*dans le module curses*), 945
ACS_BULLET (*dans le module curses*), 945
ACS_CKBOARD (*dans le module curses*), 945
ACS_DARROW (*dans le module curses*), 945
ACS_DEGREE (*dans le module curses*), 945
ACS_DIAMOND (*dans le module curses*), 945
ACS_GEQUAL (*dans le module curses*), 945
ACS_HLINE (*dans le module curses*), 945
ACS_LANTERN (*dans le module curses*), 945
ACS_LARROW (*dans le module curses*), 945
ACS_LEQUAL (*dans le module curses*), 945
ACS_LLCORNER (*dans le module curses*), 945
ACS_LRCORNER (*dans le module curses*), 946
ACS_LTEE (*dans le module curses*), 946
ACS_NEQUAL (*dans le module curses*), 946
ACS_PI (*dans le module curses*), 946
ACS_PLMINUS (*dans le module curses*), 946
ACS_PLUS (*dans le module curses*), 946
ACS_RARROW (*dans le module curses*), 946
ACS_RTEE (*dans le module curses*), 946
ACS_S1 (*dans le module curses*), 946
ACS_S3 (*dans le module curses*), 946
ACS_S7 (*dans le module curses*), 946
ACS_S9 (*dans le module curses*), 946
ACS_SBBS (*dans le module curses*), 946
ACS_SBSB (*dans le module curses*), 946
ACS_SBSS (*dans le module curses*), 946
ACS_SSBB (*dans le module curses*), 946
ACS_SSBS (*dans le module curses*), 946
ACS_SSSB (*dans le module curses*), 946
ACS_SSSS (*dans le module curses*), 947
ACS_STERLING (*dans le module curses*), 947
ACS_TTEE (*dans le module curses*), 947
ACS_UARROW (*dans le module curses*), 947
ACS_ULCORNER (*dans le module curses*), 947
ACS_URCORNER (*dans le module curses*), 947
ACS_VLINE (*dans le module curses*), 947
action (*attribut optparse.Option*), 907
Action (*classe dans argparse*), 867
ACTIONS (*attribut optparse.Option*), 919
activate_stack_trampoline() (*dans le module sys*), 1951
active_children() (*dans le module multiprocessing*), 982
active_count() (*dans le module threading*), 956
actual() (*méthode tkinter.font.Font*), 1633
Add (*classe dans ast*), 2110
add() (*dans le module operator*), 438
add() (*méthode decimal.Context*), 366
add() (*méthode frozenset*), 90
add() (*méthode graphlib.TopologicalSorter*), 331
add() (*méthode mailbox.Mailbox*), 1318
add() (*méthode mailbox.Maildir*), 1322
add() (*méthode pstats.Stats*), 1890
add() (*méthode tarfile.TarFile*), 602
add() (*méthode tkinter.ttk.Notebook*), 1647
add_alias() (*dans le module email.charset*), 1303
add_alternative() (*méthode email.message.EmailMessage*), 1257
add_argument() (*méthode argparse.ArgumentParser*), 858
add_argument_group() (*méthode argparse.ArgumentParser*), 875
add_attachment() (*méthode email.message.EmailMessage*), 1257
add_cgi_vars() (*méthode wsgiref.handlers.BaseHandler*), 1422
add_charset() (*dans le module email.charset*), 1302
add_codec() (*dans le module email.charset*), 1303
add_cookie_header() (*méthode http.cookiejar.CookieJar*), 1514
add_dll_directory() (*dans le module os*), 712
add_done_callback() (*méthode asyncio.Future*), 1132
add_done_callback() (*méthode asyncio.Task*), 1081
add_done_callback() (*méthode concurrent.futures.Future*), 1026
add_fallback() (*méthode gettext.NullTranslations*), 1555
add_flag() (*méthode mailbox.Maildir*), 1322
add_flag() (*méthode mailbox.MaildirMessage*), 1327
add_flag() (*méthode mailbox.mboxMessage*), 1329

add_flag() (*méthode mailbox.MMDFMessage*), 1333
add_folder() (*méthode mailbox.Maildir*), 1321
add_folder() (*méthode mailbox.MH*), 1324
add_get_handler() (*méthode email.contentmanager.ContentManager*), 1279
add_handler() (*méthode urllib.request.OpenerDirector*), 1433
add_header() (*méthode email.message.EmailMessage*), 1253
add_header() (*méthode email.message.Message*), 1292
add_header() (*méthode urllib.request.Request*), 1432
add_header() (*méthode wsgiref.headers.Headers*), 1418
add_history() (*dans le module readline*), 177
add_label() (*méthode mailbox.BabylMessage*), 1332
add_mutually_exclusive_group() (*méthode argparse.ArgumentParser*), 876
add_note() (*méthode BaseException*), 108
add_option() (*méthode optparse.OptionParser*), 906
add_parent() (*méthode urllib.request.BaseHandler*), 1434
add_password() (*méthode urllib.request.HTTPPasswordMgr*), 1436
add_password() (*méthode urllib.request.HTTPPasswordMgrWithPriorAuth*), 1436
add_reader() (*méthode asyncio.loop*), 1118
add_related() (*méthode email.message.EmailMessage*), 1257
add_section() (*méthode configparser.ConfigParser*), 635
add_section() (*méthode configparser.RawConfigParser*), 638
add_sequence() (*méthode mailbox.MHMessage*), 1331
add_set_handler() (*méthode email.contentmanager.ContentManager*), 1279
add_signal_handler() (*méthode asyncio.loop*), 1121
add_subparsers() (*méthode argparse.ArgumentParser*), 871
add_type() (*dans le module mimetypes*), 1337
add_type() (*méthode mimetypes.MimeTypes*), 1339
add_unredirected_header() (*méthode urllib.request.Request*), 1432
add_writer() (*méthode asyncio.loop*), 1118
addAsyncCleanup() (*méthode unittest.IsolatedAsyncioTestCase*), 1772
addaudithook() (*dans le module sys*), 1929
addch() (*méthode curses.window*), 931
addClassCleanup() (*méthode de la classe unittest.TestCase*), 1771
addCleanup() (*méthode unittest.TestCase*), 1770
addcomponent() (*méthode turtle.Shape*), 1603
addDuration() (*méthode unittest.TestResult*), 1778
addError() (*méthode unittest.TestResult*), 1778
addExpectedFailure() (*méthode unittest.TestResult*), 1778
addFailure() (*méthode unittest.TestResult*), 1778
addfile() (*méthode tarfile.TarFile*), 602
addFilter() (*méthode logging.Handler*), 763
addFilter() (*méthode logging.Logger*), 761
addHandler() (*méthode logging.Logger*), 761
addinfourl (*classe dans urllib.response*), 1442
addLevelName() (*dans le module logging*), 771
addModuleCleanup() (*dans le module unittest*), 1782
addnstr() (*méthode curses.window*), 931
AddPackagePath() (*dans le module modulefinder*), 2062
addr_spec (*attribut email.headerregistry.Address*), 1278
address (*attribut email.headerregistry.SingleAddressHeader*), 1275
address (*attribut multiprocessing.connection.Listener*), 1002
address (*attribut multiprocessing.managers.BaseManager*), 992
Address (*classe dans email.headerregistry*), 1277
address_exclude() (*méthode ipaddress.IPv4Network*), 1542
address_exclude() (*méthode ipaddress.IPv6Network*), 1544
address_family (*attribut socketserver.BaseServer*), 1496
address_string() (*méthode http.server.BaseHTTPRequestHandler*), 1506
addresses (*attribut email.headerregistry.AddressHeader*), 1275
addresses (*attribut email.headerregistry.Group*), 1278
AddressHeader (*classe dans email.headerregistry*), 1275
addressof() (*dans le module ctypes*), 839
AddressValueError, 1547
addshape() (*dans le module turtle*), 1600
addsitedir() (*dans le module site*), 2051
addSkip() (*méthode unittest.TestResult*), 1778
addstr() (*méthode curses.window*), 931
addSubTest() (*méthode unittest.TestResult*), 1778
addSuccess() (*méthode unittest.TestResult*), 1778
addTest() (*méthode unittest.TestSuite*), 1773
addTests() (*méthode unittest.TestSuite*), 1773
addTypeEqualityFunc() (*méthode unittest.TestCase*), 1768
addUnexpectedSuccess() (*méthode unittest.TestResult*), 1778
adjust_int_max_str_digits() (*dans le module test.support*), 1855
adjusted() (*méthode decimal.Decimal*), 357
adler32() (*dans le module zlib*), 567
AF_ALG (*dans le module socket*), 1169
AF_CAN (*dans le module socket*), 1167
AF_DIVERT (*dans le module socket*), 1168
AF_HYPERV (*dans le module socket*), 1170

AF_INET (*dans le module socket*), 1166
AF_INET6 (*dans le module socket*), 1166
AF_LINK (*dans le module socket*), 1169
AF_PACKET (*dans le module socket*), 1168
AF_QIPCRTR (*dans le module socket*), 1170
AF_RDS (*dans le module socket*), 1169
AF_UNIX (*dans le module socket*), 1166
AF_UNSPEC (*dans le module socket*), 1166
AF_VSOCK (*dans le module socket*), 1169
aifc
    module, 2221
aiter()
    built-in function, 8
alarm() (*dans le module signal*), 1238
ALERT_DESCRIPTION_HANDSHAKE_FAILURE (*dans le module ssl*), 1201
ALERT_DESCRIPTION_INTERNAL_ERROR (*dans le module ssl*), 1201
AlertDescription (*classe dans ssl*), 1201
algorithm (*attribut sys.hash_info*), 1943
algorithms_available (*dans le module hashlib*), 647
algorithms_guaranteed (*dans le module hashlib*), 647
Alias
    Generic, 96
alias (*classe dans ast*), 2118
alias (*pdb command*), 1884
alias de type, **2244**
alignment() (*dans le module ctypes*), 839
alive (*attribut weakref.finalize*), 295
all()
    built-in function, 8
ALL_COMPLETED (*dans le module asyncio*), 1077
ALL_COMPLETED (*dans le module concurrent.futures*), 1027
all_errors (*dans le module ftplib*), 1471
all_features (*dans le module xml.sax.handler*), 1394
all_frames (*attribut tracemalloc.Filter*), 1908
all_properties (*dans le module xml.sax.handler*), 1394
all_suffixes() (*dans le module importlib.machinery*), 2075
all_tasks() (*dans le module asyncio*), 1080
allocate_lock() (*dans le module _thread*), 1057
allow_reuse_address (*attribut socketserver.BaseServer*), 1497
allowed_domains() (*méthode http.cookiejar.DefaultCookiePolicy*), 1518
alt() (*dans le module curses.ascii*), 953
ALT_DIGITS (*dans le module locale*), 1565
altsep (*dans le module os*), 728
altzone (*dans le module time*), 755
ALWAYS_EQ (*dans le module test.support*), 1849
ALWAYS_TYPED_ACTIONS (*attribut optparse.Option*), 919
AmbiguousOptionError, 920
AMPER (*dans le module token*), 2144
AMPEREQUAL (*dans le module token*), 2145
anchor (*attribut pathlib.PurePath*), 451
Anchor (*classe dans importlib.resources*), 2085
and
    operator, 37, 38
And (*classe dans ast*), 2111
and_() (*dans le module operator*), 438
android_ver() (*dans le module platform*), 804
anext()
    built-in function, 8
AnnAssign (*classe dans ast*), 2116
--annotate
    option de ligne de commande pickletools, 2182
annotate function, **2229**
Annotated (*dans le module typing*), 1690
annotation, **2229**
    type annotation; type hint, 96
annotation (*attribut inspect.Parameter*), 2034
ANNOTATION (*attribut symtable.SymbolTableType*), 2140
annotation de fonction, **2235**
annotation de variable, **2244**
annotationlib
    module, 2045
annotations_to_string() (*dans le module annotationlib*), 2047
answer_challenge() (*dans le module multiprocessing.connection*), 1001
anticipate_failure() (*dans le module test.support*), 1852
Any (*dans le module typing*), 1683
ANY (*dans le module unittest.mock*), 1817
any()
    built-in function, 9
ANY_CONTIGUOUS (*attribut inspect.BufferFlags*), 2044
AnyStr (*dans le module typing*), 1683
API provisoire, **2241**
api_version (*dans le module sys*), 1954
apilevel (*dans le module sqlite3*), 541
apop() (*méthode poplib.POP3*), 1472
appelable (*callable*), **2231**
append() (*méthode array.array*), 290
append() (*méthode collections.deque*), 265
append() (*méthode email.header.Header*), 1299
append() (*méthode imaplib.IMAP4*), 1476
append() (*méthode xml.etree.ElementTree.Element*), 1367
append() (*sequence method*), 49
append_history_file() (*dans le module readline*), 177
appendChild() (*méthode xml.dom.Node*), 1377
appendleft() (*méthode collections.deque*), 265
AppleFrameworkLoader (*classe dans importlib.machinery*), 2079
application_uri() (*dans le module wsgiref.util*), 1416
apply() (*méthode multiprocessing.pool.Pool*), 998

apply_async() (*méthode multiprocessing.pool.Pool*), 998
apply_defaults() (*méthode inspect.BoundArguments*), 2036
APRIL (*dans le module calendar*), 256
architecture() (*dans le module platform*), 800
archive (*attribut zipimport.zipimporter*), 2059
AREGTYPE (*dans le module tarfile*), 598
aRepr (*dans le module reprlib*), 313
arg (*classe dans ast*), 2131
argparse
    module, 849
args (*attribut BaseException*), 108
args (*attribut functools.partial*), 437
args (*attribut inspect.BoundArguments*), 2036
args (*attribut subprocess.CompletedProcess*), 1030
args (*attribut subprocess.Popen*), 1039
args (*attribut typing.ParamSpec*), 1700
args (*pdb command*), 1882
args_from_interpreter_flags() (*dans le module test.support*), 1850
argtypes (*attribut ctypes._CFuncPtr*), 836
argument, **2229**
argument nommé, **2238**
argument positionnel, **2241**
ArgumentDefaultsHelpFormatter (*classe dans argparse*), 853
ArgumentError, 848, 880
ArgumentParser (*classe dans argparse*), 850
arguments (*attribut inspect.BoundArguments*), 2035
arguments (*classe dans ast*), 2130
ArgumentTypeError, 880
argv (*dans le module sys*), 1930
arithmetic, 39
ArithmeticError, 109
array
    module, 65, 289
array (*classe dans array*), 290
Array (*classe dans ctypes*), 848
ARRAY() (*dans le module ctypes*), 848
Array() (*dans le module multiprocessing*), 988
Array() (*dans le module multiprocessing.sharedctypes*), 989
Array() (*méthode multiprocessing.managers.SyncManager*), 993
arrays, 289
arraysize (*attribut sqlite3.Cursor*), 554
arrêt de l'interpréteur, **2237**
as_bytes() (*méthode email.message.EmailMessage*), 1251
as_bytes() (*méthode email.message.Message*), 1289
as_completed() (*dans le module asyncio*), 1077
as_completed() (*dans le module concurrent.futures*), 1027
as_file() (*dans le module importlib.resources*), 2085
as_integer_ratio(), 380
as_integer_ratio() (*méthode decimal.Decimal*), 357
as_integer_ratio() (*méthode float*), 43
as_integer_ratio() (*méthode fractions.Fraction*), 381
as_integer_ratio() (*méthode int*), 43
as_posix() (*méthode pathlib.PurePath*), 453
as_string() (*méthode email.message.EmailMessage*), 1251
as_string() (*méthode email.message.Message*), 1288
as_tuple() (*méthode decimal.Decimal*), 357
as_uri() (*méthode pathlib.Path*), 458
ASCII (*dans le module re*), 140
ascii()
    built-in function, 9
ascii() (*dans le module curses.ascii*), 952
ascii_letters (*dans le module string*), 121
ascii_lowercase (*dans le module string*), 121
ascii_uppercase (*dans le module string*), 121
asctime() (*dans le module time*), 744
asdict() (*dans le module dataclasses*), 1982
asin() (*dans le module cmath*), 349
asin() (*dans le module math*), 345
asinh() (*dans le module cmath*), 349
asinh() (*dans le module math*), 345
askcolor() (*dans le module tkinter.colorchooser*), 1632
askdirectory() (*dans le module tkinter.filedialog*), 1635
askfloat() (*dans le module tkinter.simpledialog*), 1634
askinteger() (*dans le module tkinter.simpledialog*), 1634
askokcancel() (*dans le module tkinter.messagebox*), 1638
askopenfile() (*dans le module tkinter.filedialog*), 1635
askopenfilename() (*dans le module tkinter.filedialog*), 1635
askopenfilenames() (*dans le module tkinter.filedialog*), 1635
askopenfiles() (*dans le module tkinter.filedialog*), 1635
askquestion() (*dans le module tkinter.messagebox*), 1638
askretrycancel() (*dans le module tkinter.messagebox*), 1638
asksaveasfile() (*dans le module tkinter.filedialog*), 1635
asksaveasfilename() (*dans le module tkinter.filedialog*), 1635
askstring() (*dans le module tkinter.simpledialog*), 1634
askyesno() (*dans le module tkinter.messagebox*), 1638
askyesnocancel() (*dans le module tkinter.messagebox*), 1638
assert
    statement, 109
Assert (*classe dans ast*), 2117
assert_any_await() (*méthode unittest.mock.AsyncMock*), 1798
assert_any_call() (*méthode unittest.mock.Mock*),

1788
assert_awaited() (*méthode unittest.mock.AsyncMock*), 1796
assert_awaited_once() (*méthode unittest.mock.AsyncMock*), 1797
assert_awaited_once_with() (*méthode unittest.mock.AsyncMock*), 1797
assert_awaited_with() (*méthode unittest.mock.AsyncMock*), 1797
assert_called() (*méthode unittest.mock.Mock*), 1787
assert_called_once() (*méthode unittest.mock.Mock*), 1787
assert_called_once_with() (*méthode unittest.mock.Mock*), 1787
assert_called_with() (*méthode unittest.mock.Mock*), 1787
assert_has_awaits() (*méthode unittest.mock.AsyncMock*), 1798
assert_has_calls() (*méthode unittest.mock.Mock*), 1788
assert_never() (*dans le module typing*), 1710
assert_not_awaited() (*méthode unittest.mock.AsyncMock*), 1798
assert_not_called() (*méthode unittest.mock.Mock*), 1788
assert_python_failure() (*dans le module test.support.script_helper*), 1857
assert_python_ok() (*dans le module test.support.script_helper*), 1857
assert_type() (*dans le module typing*), 1710
assertAlmostEqual() (*méthode unittest.TestCase*), 1767
assertCountEqual() (*méthode unittest.TestCase*), 1768
assertDictEqual() (*méthode unittest.TestCase*), 1769
assertEndsWith() (*méthode unittest.TestCase*), 1768
assertEqual() (*méthode unittest.TestCase*), 1763
assertFalse() (*méthode unittest.TestCase*), 1763
assertGreater() (*méthode unittest.TestCase*), 1767
assertGreaterEqual() (*méthode unittest.TestCase*), 1767
assertHasAttr() (*méthode unittest.TestCase*), 1768
assertIn() (*méthode unittest.TestCase*), 1763
assertInBytecode() (*méthode test.support.bytecode_helper.BytecodeTestCase*), 1858
AssertionError, 109
assertIs() (*méthode unittest.TestCase*), 1763
assertIsInstance() (*méthode unittest.TestCase*), 1764
assertIsNone() (*méthode unittest.TestCase*), 1763
assertIsNot() (*méthode unittest.TestCase*), 1763
assertIsNotNone() (*méthode unittest.TestCase*), 1763
assertIsSubclass() (*méthode unittest.TestCase*), 1764
assertLess() (*méthode unittest.TestCase*), 1767
assertLessEqual() (*méthode unittest.TestCase*), 1767
assertListEqual() (*méthode unittest.TestCase*), 1769
assertLogs() (*méthode unittest.TestCase*), 1766
assertMultiLineEqual() (*méthode unittest.TestCase*), 1769
assertNoLogs() (*méthode unittest.TestCase*), 1766
assertNotAlmostEqual() (*méthode unittest.TestCase*), 1767
assertNotEndsWith() (*méthode unittest.TestCase*), 1768
assertNotEqual() (*méthode unittest.TestCase*), 1763
assertNotHasAttr() (*méthode unittest.TestCase*), 1768
assertNotIn() (*méthode unittest.TestCase*), 1763
assertNotInBytecode() (*méthode test.support.bytecode_helper.BytecodeTestCase*), 1858
assertNotIsInstance() (*méthode unittest.TestCase*), 1764
assertNotIsSubclass() (*méthode unittest.TestCase*), 1764
assertNotRegex() (*méthode unittest.TestCase*), 1768
assertNotStartsWith() (*méthode unittest.TestCase*), 1768
assertRaises() (*méthode unittest.TestCase*), 1764
assertRaisesRegex() (*méthode unittest.TestCase*), 1765
assertRegex() (*méthode unittest.TestCase*), 1768
assertSequenceEqual() (*méthode unittest.TestCase*), 1769
assertSetEqual() (*méthode unittest.TestCase*), 1769
assertStartsWith() (*méthode unittest.TestCase*), 1768
assertTrue() (*méthode unittest.TestCase*), 1763
assertTupleEqual() (*méthode unittest.TestCase*), 1769
assertWarns() (*méthode unittest.TestCase*), 1765
assertWarnsRegex() (*méthode unittest.TestCase*), 1766
Assign (*classe dans ast*), 2115
assignment
    slice, 49
    subscript, 49
ast
    module, 2101
AST (*classe dans ast*), 2104
astimezone() (*méthode datetime.datetime*), 223
astuple() (*dans le module dataclasses*), 1983
AsyncContextDecorator (*classe dans contextlib*), 1996
AsyncContextManager (*classe dans typing*), 1723
asynccontextmanager() (*dans le module contextlib*), 1990
AsyncExitStack (*classe dans contextlib*), 1998
AsyncFor (*classe dans ast*), 2133

AsyncFunctionDef (*classe dans ast*), 2133
AsyncGenerator (*classe dans collections.abc*), 280
AsyncGenerator (*classe dans typing*), 1722
AsyncGeneratorType (*dans le module types*), 301
asynchat
    module, 2221
asyncio
    module, 1061
asyncio.subprocess.DEVNULL (*variable de base*), 1099
asyncio.subprocess.PIPE (*variable de base*), 1099
asyncio.subprocess.Process (*classe de base*), 1099
asyncio.subprocess.STDOUT (*variable de base*), 1099
AsyncIterable (*classe dans collections.abc*), 280
AsyncIterable (*classe dans typing*), 1722
AsyncIterator (*classe dans collections.abc*), 280
AsyncIterator (*classe dans typing*), 1722
AsyncMock (*classe dans unittest.mock*), 1796
asyncore
    module, 2221
AsyncResult (*classe dans multiprocessing.pool*), 1000
asyncSetUp() (*méthode unittest.IsolatedAsyncioTestCase*), 1771
asyncTearDown() (*méthode unittest.IsolatedAsyncioTestCase*), 1772
AsyncWith (*classe dans ast*), 2133
AT (*dans le module token*), 2146
at_eof() (*méthode asyncio.StreamReader*), 1087
atan() (*dans le module cmath*), 349
atan() (*dans le module math*), 345
atan2() (*dans le module math*), 345
atanh() (*dans le module cmath*), 349
atanh() (*dans le module math*), 345
ATEQUAL (*dans le module token*), 2146
atexit
    module, 2009
atexit (*attribut weakref.finalize*), 295
atof() (*dans le module locale*), 1567
atoi() (*dans le module locale*), 1567
attach() (*méthode email.message.Message*), 1289
attach_mock() (*méthode unittest.mock.Mock*), 1789
attempted (*attribut doctest.TestResults*), 1745
attendable (*awaitable*), **2230**
AttlistDeclHandler() (*méthode xml.parsers.expat.xmlparser*), 1406
attrgetter() (*dans le module operator*), 440
attrib (*attribut xml.etree.ElementTree.Element*), 1366
attribut, **2230**
Attribute (*classe dans ast*), 2112
AttributeError, 109
attributes (*attribut xml.dom.Node*), 1376
AttributesImpl (*classe dans xml.sax.xmlreader*), 1399
AttributesNSImpl (*classe dans xml.sax.xmlreader*), 1399
attroff() (*méthode curses.window*), 931
attron() (*méthode curses.window*), 931
attrset() (*méthode curses.window*), 931
audioop
    module, 2222
audit events, 1865
audit() (*dans le module sys*), 1930
auditing, 1930
AugAssign (*classe dans ast*), 2116
AUGUST (*dans le module calendar*), 256
auth() (*méthode ftplib.FTP_TLS*), 1470
auth() (*méthode smtplib.SMTP*), 1485
authenticate() (*méthode imaplib.IMAP4*), 1476
AuthenticationError, 979
authenticators() (*méthode netrc.netrc*), 642
authkey (*attribut multiprocessing.Process*), 978
auto (*classe dans enum*), 329
autocommit (*attribut sqlite3.Connection*), 551
autorange() (*méthode timeit.Timer*), 1895
available_timezones() (*dans le module zoneinfo*), 249
avoids_symlink_attacks (*attribut shutil.rmtree*), 500
Await (*classe dans ast*), 2133
await_args (*attribut unittest.mock.AsyncMock*), 1799
await_args_list (*attribut unittest.mock.AsyncMock*), 1799
await_count (*attribut unittest.mock.AsyncMock*), 1798
Awaitable (*classe dans collections.abc*), 280
Awaitable (*classe dans typing*), 1722

B

-b
    option de ligne de commande compileall, 2156
    option de ligne de commande unittest, 1754
b2a_base64() (*dans le module binascii*), 1343
b2a_hex() (*dans le module binascii*), 1344
b2a_qp() (*dans le module binascii*), 1343
b2a_uu() (*dans le module binascii*), 1343
b16decode() (*dans le module base64*), 1341
b16encode() (*dans le module base64*), 1340
b32decode() (*dans le module base64*), 1340
b32encode() (*dans le module base64*), 1340
b32hexdecode() (*dans le module base64*), 1340
b32hexencode() (*dans le module base64*), 1340
b64decode() (*dans le module base64*), 1340
b64encode() (*dans le module base64*), 1339
b85decode() (*dans le module base64*), 1341
b85encode() (*dans le module base64*), 1341
Babyl (*classe dans mailbox*), 1325
BabylMessage (*classe dans mailbox*), 1331
back() (*dans le module turtle*), 1578
backend (*dans le module readline*), 176
backslashreplace
    error handler's name, 192
backslashreplace_errors() (*dans le module codecs*), 193

backup() (*méthode sqlite3.Connection*), 549
backward() (*dans le module turtle*), 1578
BadGzipFile, 571
BadOptionError, 920
BadStatusLine, 1459
BadZipFile, 584
BadZipfile, 584
Barrier (*classe dans asyncio*), 1096
Barrier (*classe dans multiprocessing*), 986
Barrier (*classe dans threading*), 968
Barrier() (*méthode multiprocessing.managers.SyncManager*), 993
base64
    encoding, 1339
    module, 1339, 1343
base_exec_prefix (*dans le module sys*), 1930
base_prefix (*dans le module sys*), 1931
BaseCGIHandler (*classe dans wsgiref.handlers*), 1421
BaseCookie (*classe dans http.cookies*), 1509
BaseException, 108
BaseExceptionGroup, 117
BaseHandler (*classe dans urllib.request*), 1429
BaseHandler (*classe dans wsgiref.handlers*), 1422
BaseHeader (*classe dans email.headerregistry*), 1273
BaseHTTPRequestHandler (*classe dans http.server*), 1503
BaseManager (*classe dans multiprocessing.managers*), 991
basename() (*dans le module os.path*), 473
BaseProtocol (*classe dans asyncio*), 1139
BaseProxy (*classe dans multiprocessing.managers*), 997
BaseRequestHandler (*classe dans socketserver*), 1498
BaseRotatingHandler (*classe dans logging.handlers*), 788
BaseSelector (*classe dans selectors*), 1232
BaseServer (*classe dans socketserver*), 1496
BaseTransport (*classe dans asyncio*), 1135
basicConfig() (*dans le module logging*), 772
BasicContext (*dans le module decimal*), 364
BasicInterpolation (*classe dans configparser*), 625
batched() (*dans le module itertools*), 411
baudrate() (*dans le module curses*), 924
bbox() (*méthode tkinter.ttk.Treeview*), 1651
BDADDR_ANY (*dans le module socket*), 1169
BDADDR_LOCAL (*dans le module socket*), 1169
bdb
    module, 1869, 1876
Bdb (*classe dans bdb*), 1870
BdbQuit, 1869
BDFL, **2230**
beep() (*dans le module curses*), 924
Beep() (*dans le module winsound*), 2194
begin_fill() (*dans le module turtle*), 1588
begin_poly() (*dans le module turtle*), 1593
BEL (*dans le module curses.ascii*), 950
below() (*méthode curses.panel.Panel*), 953
BELOW_NORMAL_PRIORITY_CLASS (*dans le module subprocess*), 1041
Benchmarking, 1893
benchmarking, 747, 752
--best
    option de ligne de commande gzip, 574
betavariate() (*dans le module random*), 387
bgcolor() (*dans le module turtle*), 1595
bgpic() (*dans le module turtle*), 1595
bidirectional() (*dans le module unicodedata*), 172
bigaddrspacetest() (*dans le module test.support*), 1853
BigEndianStructure (*classe dans ctypes*), 846
BigEndianUnion (*classe dans ctypes*), 846
bigmemtest() (*dans le module test.support*), 1853
bin()
    built-in function, 9
binary
    data, packing, 181
    literals, 39
Binary (*classe dans xmlrpc.client*), 1524
binary semaphores, 1056
BINARY_OP (*opcode*), 2167
BINARY_SLICE (*opcode*), 2167
BinaryIO (*classe dans typing*), 1710
binascii
    module, 1343
bind (*widgets*), 1630
bind() (*méthode inspect.Signature*), 2033
bind() (*méthode socket.socket*), 1178
bind_partial() (*méthode inspect.Signature*), 2033
bind_port() (*dans le module test.support.socket_helper*), 1856
bind_textdomain_codeset() (*dans le module locale*), 1569
bind_unix_socket() (*dans le module test.support.socket_helper*), 1856
bindtextdomain() (*dans le module gettext*), 1553
bindtextdomain() (*dans le module locale*), 1569
binomialvariate() (*dans le module random*), 386
BinOp (*classe dans ast*), 2110
bisect
    module, 286
bisect() (*dans le module bisect*), 286
bisect_left() (*dans le module bisect*), 286
bisect_right() (*dans le module bisect*), 286
bit_count() (*méthode int*), 41
bit_length() (*méthode int*), 41
BitAnd (*classe dans ast*), 2110
BitOr (*classe dans ast*), 2110
bits_per_digit (*attribut sys.int_info*), 1944
bitwise
    operations, 40
BitXor (*classe dans ast*), 2110
bk() (*dans le module turtle*), 1578
bkgd() (*méthode curses.window*), 931
bkgdset() (*méthode curses.window*), 931
blake2b() (*dans le module hashlib*), 650
blake2b, blake2s, 650

blake2b.MAX_DIGEST_SIZE (*dans le module hashlib*), 651
blake2b.MAX_KEY_SIZE (*dans le module hashlib*), 651
blake2b.PERSON_SIZE (*dans le module hashlib*), 651
blake2b.SALT_SIZE (*dans le module hashlib*), 651
blake2s() (*dans le module hashlib*), 650
blake2s.MAX_DIGEST_SIZE (*dans le module hashlib*), 651
blake2s.MAX_KEY_SIZE (*dans le module hashlib*), 651
blake2s.PERSON_SIZE (*dans le module hashlib*), 651
blake2s.SALT_SIZE (*dans le module hashlib*), 651
BLKTYPE (*dans le module tarfile*), 598
Blob (*classe dans sqlite3*), 555
blobopen() (*méthode sqlite3.Connection*), 543
block_on_close (*attribut socketserver.ThreadingMixIn*), 1495
block_size (*attribut hmac.HMAC*), 658
blocked_domains() (*méthode http.cookiejar.DefaultCookiePolicy*), 1517
BlockingIOError, 115, 732
blocksize (*attribut http.client.HTTPConnection*), 1461
body() (*méthode tkinter.simpledialog.Dialog*), 1634
body_encode() (*méthode email.charset.Charset*), 1302
body_encoding (*attribut email.charset.Charset*), 1301
body_line_iterator() (*dans le module email.iterators*), 1306
BOLD (*dans le module tkinter.font*), 1633
BOM (*dans le module codecs*), 191
BOM_BE (*dans le module codecs*), 191
BOM_LE (*dans le module codecs*), 191
BOM_UTF8 (*dans le module codecs*), 191
BOM_UTF16 (*dans le module codecs*), 191
BOM_UTF16_BE (*dans le module codecs*), 191
BOM_UTF16_LE (*dans le module codecs*), 191
BOM_UTF32 (*dans le module codecs*), 191
BOM_UTF32_BE (*dans le module codecs*), 191
BOM_UTF32_LE (*dans le module codecs*), 191
bool (*classe de base*), 9
Boolean
    object, 38
    operations, 37
    type, 9
    values, 46
BOOLEAN_STATES (*attribut configparser.ConfigParser*), 630
BooleanOptionalAction (*classe dans argparse*), 860
BoolOp (*classe dans ast*), 2110
bootstrap() (*dans le module ensurepip*), 1914
border() (*méthode curses.window*), 932
bottom() (*méthode curses.panel.Panel*), 953
bottom_panel() (*dans le module curses.panel*), 953
BoundArguments (*classe dans inspect*), 2035
BoundaryError, 1272
BoundedSemaphore (*classe dans asyncio*), 1096
BoundedSemaphore (*classe dans multiprocessing*), 986
BoundedSemaphore (*classe dans threading*), 966
BoundedSemaphore() (*méthode multiprocessing.managers.SyncManager*), 993
box() (*méthode curses.window*), 932
bpbynumber (*attribut bdb.Breakpoint*), 1870
bpformat() (*méthode bdb.Breakpoint*), 1869
bplist (*attribut bdb.Breakpoint*), 1870
bpprint() (*méthode bdb.Breakpoint*), 1870
BRANCH_LEFT (*monitoring event*), 1956
BRANCH_RIGHT (*monitoring event*), 1956
Break (*classe dans ast*), 2120
break (*pdb command*), 1880
break_anywhere() (*méthode bdb.Bdb*), 1871
break_here() (*méthode bdb.Bdb*), 1871
break_long_words (*attribut textwrap.TextWrapper*), 171
break_on_hyphens (*attribut textwrap.TextWrapper*), 171
Breakpoint (*classe dans bdb*), 1869
breakpoint()
    built-in function, 9
breakpointhook() (*dans le module sys*), 1932
breakpoints, 1663
broadcast_address (*attribut ipaddress.IPv4Network*), 1541
broadcast_address (*attribut ipaddress.IPv6Network*), 1544
broken (*attribut asyncio.Barrier*), 1097
broken (*attribut threading.Barrier*), 968
BrokenBarrierError, 968, 1097
BrokenExecutor, 1027
BrokenInterpreterPool, 1028
BrokenPipeError, 115
BrokenProcessPool, 1028
BrokenThreadPool, 1028
BROWSER, 1413, 1414
BS (*dans le module curses.ascii*), 950
BsdDbShelf (*classe dans shelve*), 528
buf (*attribut multiprocessing.shared_memory.SharedMemory*), 1015
--buffer
    option de ligne de commande unittest, 1754
buffer (*attribut io.TextIOBase*), 740
buffer (*attribut unittest.TestResult*), 1777
Buffer (*classe dans collections.abc*), 280
buffer protocol
    binary sequence types, 64
    str (*built-in class*), 54
buffer size, I/O, 25
buffer_info() (*méthode array.array*), 290
buffer_size (*attribut xml.parsers.expat.xmlparser*), 1405
buffer_text (*attribut xml.parsers.expat.xmlparser*), 1405
buffer_updated() (*méthode asyncio.BufferedProtocol*), 1141
buffer_used (*attribut xml.parsers.expat.xmlparser*), 1405

BufferedIOBase (*classe dans io*), 736
BufferedProtocol (*classe dans asyncio*), 1139
BufferedRandom (*classe dans io*), 739
BufferedReader (*classe dans io*), 738
BufferedRWPair (*classe dans io*), 739
BufferedWriter (*classe dans io*), 739
BufferError, 109
BufferFlags (*classe dans inspect*), 2044
BufferingFormatter (*classe dans logging*), 765
BufferingHandler (*classe dans logging.handlers*), 796
BufferTooShort, 979
BUILD_LIST (*opcode*), 2172
BUILD_MAP (*opcode*), 2172
build_opener() (*dans le module urllib.request*), 1427
BUILD_SET (*opcode*), 2172
BUILD_SLICE (*opcode*), 2177
BUILD_STRING (*opcode*), 2172
BUILD_TUPLE (*opcode*), 2172
built-in
    types, 37
built-in function
    __import__(), 33
    abs(), 8
    aiter(), 8
    all(), 8
    anext(), 8
    any(), 9
    ascii(), 9
    bin(), 9
    breakpoint(), 9
    callable(), 10
    chr(), 10
    classmethod(), 10
    compile, 103, 302
    compile(), 11
    complex, 39
    delattr(), 13
    dir(), 13
    divmod(), 14
    enumerate(), 14
    eval, 103, 308, 309
    eval(), 14
    exec, 15, 103
    exec(), 15
    filter(), 16
    float, 39
    format(), 17
    getattr(), 18
    globals(), 18
    hasattr(), 18
    hash, 48
    hash(), 18
    help(), 18
    hex(), 18
    id(), 19
    input(), 19
    int, 39
    isinstance(), 20
    issubclass(), 20
    iter(), 21
    len, 47, 90
    len(), 21
    locals(), 21
    map(), 22
    max, 47
    max(), 22
    min, 47
    min(), 22
    multiprocessing.Manager(), 991
    next(), 22
    oct(), 23
    open(), 23
    ord(), 26
    pow(), 26
    print(), 26
    property.deleter(), 27
    property.getter(), 27
    property.setter(), 27
    repr(), 28
    reversed(), 28
    round(), 28
    setattr(), 29
    slice, 2177
    sorted(), 29
    staticmethod(), 30
    sum(), 30
    type, 103
    vars(), 32
    zip(), 32
builtin_module_names (*dans le module sys*), 1931
BuiltinFunctionType (*dans le module types*), 302
BuiltinImporter (*classe dans importlib.machinery*), 2075
BuiltinMethodType (*dans le module types*), 302
builtins
    module, 33, 1965
burst() (*méthode imaplib.IMAP4.Idler*), 1477
busy_retry() (*dans le module test.support*), 1849
BUTTON_ALT (*dans le module curses*), 947
BUTTON_CTRL (*dans le module curses*), 947
BUTTON_SHIFT (*dans le module curses*), 947
buttonbox() (*méthode tkinter.simpledialog.Dialog*), 1634
BUTTONn_CLICKED (*dans le module curses*), 947
BUTTONn_DOUBLE_CLICKED (*dans le module curses*), 947
BUTTONn_PRESSED (*dans le module curses*), 947
BUTTONn_RELEASED (*dans le module curses*), 947
BUTTONn_TRIPLE_CLICKED (*dans le module curses*), 947
bye() (*dans le module turtle*), 1601
byref() (*dans le module ctypes*), 839
bytearray
    formatting, 79
    interpolation, 79

methods, 67
object, 49, 65, 66
bytearray (*classe de base*), 66
byte-code
file, 2154
Bytecode (*classe dans dis*), 2161
BYTECODE_SUFFIXES (*dans le module importlib.machinery*), 2074
Bytecode.codeobj (*dans le module dis*), 2161
Bytecode.first_line (*dans le module dis*), 2161
BytecodeTestCase (*classe dans test.support.bytecode_helper*), 1858
byteorder (*dans le module sys*), 1931
bytes
formatting, 79
interpolation, 79
methods, 67
object, 65
str (*built-in class*), 54
bytes (*attribut uuid.UUID*), 1489
bytes (*classe de base*), 65
bytes_le (*attribut uuid.UUID*), 1489
bytes_warning (*attribut sys.flags*), 1936
BytesFeedParser (*classe dans email.parser*), 1259
BytesGenerator (*classe dans email.generator*), 1262
BytesHeaderParser (*classe dans email.parser*), 1260
BytesIO (*classe dans io*), 738
BytesParser (*classe dans email.parser*), 1260
byteswap() (*méthode array.array*), 290
BytesWarning, 117
bz2
module, 574
BZ2Compressor (*classe dans bz2*), 576
BZ2Decompressor (*classe dans bz2*), 576
BZ2File (*classe dans bz2*), 575

C

C
language, 38
structures, 181
-C
option de ligne de commande dis, 2160
option de ligne de commande trace, 1899
-c
option de ligne de commande calendar, 258
option de ligne de commande random, 392
option de ligne de commande tarfile, 609
option de ligne de commande trace, 1899
option de ligne de commande unittest, 1754
option de ligne de commande zipapp, 1925
option de ligne de commande zipfile, 595
C14NWriterTarget (*classe dans xml.etree.ElementTree*), 1371
c_bool (*classe dans ctypes*), 845
c_byte (*classe dans ctypes*), 843
c_char (*classe dans ctypes*), 843
c_char_p (*classe dans ctypes*), 843
C_CONTIGUOUS (*attribut inspect.BufferFlags*), 2044
c_contiguous (*attribut memoryview*), 88
c_double (*classe dans ctypes*), 843
c_double_complex (*classe dans ctypes*), 844
c_float (*classe dans ctypes*), 843
c_float_complex (*classe dans ctypes*), 844
c_int (*classe dans ctypes*), 844
c_int8 (*classe dans ctypes*), 844
c_int16 (*classe dans ctypes*), 844
c_int32 (*classe dans ctypes*), 844
c_int64 (*classe dans ctypes*), 844
c_long (*classe dans ctypes*), 844
c_longdouble (*classe dans ctypes*), 843
c_longdouble_complex (*classe dans ctypes*), 844
c_longlong (*classe dans ctypes*), 844
C_RAISE (*monitoring event*), 1956
C_RETURN (*monitoring event*), 1956
c_short (*classe dans ctypes*), 844
c_size_t (*classe dans ctypes*), 844
c_ssize_t (*classe dans ctypes*), 844
c_time_t (*classe dans ctypes*), 844
c_ubyte (*classe dans ctypes*), 844
c_uint (*classe dans ctypes*), 845
c_uint8 (*classe dans ctypes*), 845
c_uint16 (*classe dans ctypes*), 845
c_uint32 (*classe dans ctypes*), 845
c_uint64 (*classe dans ctypes*), 845
c_ulong (*classe dans ctypes*), 845
c_ulonglong (*classe dans ctypes*), 845
c_ushort (*classe dans ctypes*), 845
c_void_p (*classe dans ctypes*), 845
c_wchar (*classe dans ctypes*), 845
c_wchar_p (*classe dans ctypes*), 845
CACHE (*opcode*), 2166
cache() (*dans le module functools*), 426
cache_from_source() (*dans le module importlib.util*), 2080
cached (*attribut importlib.machinery.ModuleSpec*), 2078
cached_property() (*dans le module functools*), 427
CacheFTPHandler (*classe dans urllib.request*), 1431
calcobjsize() (*dans le module test.support*), 1852
calcsize() (*dans le module struct*), 182
calcvobjsize() (*dans le module test.support*), 1852
calendar
module, 250
Calendar (*classe dans calendar*), 250
calendar() (*dans le module calendar*), 255
Call (*classe dans ast*), 2111
CALL (*monitoring event*), 1956

CALL (*opcode*), 2176
call() (*dans le module operator*), 440
call() (*dans le module subprocess*), 1042
call() (*dans le module unittest.mock*), 1816
call_annotate_function() (*dans le module annotationlib*), 2047
call_args (*attribut unittest.mock.Mock*), 1792
call_args_list (*attribut unittest.mock.Mock*), 1792
call_at() (*méthode asyncio.loop*), 1111
call_count (*attribut unittest.mock.Mock*), 1790
call_evaluate_function() (*dans le module annotationlib*), 2047
call_exception_handler() (*méthode asyncio.loop*), 1123
CALL_FUNCTION_EX (*opcode*), 2176
CALL_INTRINSIC_1 (*opcode*), 2179
CALL_INTRINSIC_2 (*opcode*), 2179
CALL_KW (*opcode*), 2176
call_later() (*méthode asyncio.loop*), 1111
call_list() (*méthode unittest.mock.call*), 1816
call_soon() (*méthode asyncio.loop*), 1110
call_soon_threadsafe() (*méthode asyncio.loop*), 1110
call_tracing() (*dans le module sys*), 1931
Callable (*classe dans collections.abc*), 279
Callable (*dans le module typing*), 1723
callable()
    built-in function, 10
CallableProxyType (*dans le module weakref*), 296
callback (*attribut optparse.Option*), 908
callback() (*méthode contextlib.ExitStack*), 1997
callback_args (*attribut optparse.Option*), 908
callback_kwargs (*attribut optparse.Option*), 908
callbacks (*dans le module gc*), 2025
called (*attribut unittest.mock.Mock*), 1790
CalledProcessError, 1031
CAN (*dans le module curses.ascii*), 951
CAN_BCM (*dans le module socket*), 1167
can_change_color() (*dans le module curses*), 924
can_fetch() (*méthode urllib.robotparser.RobotFileParser*), 1453
CAN_ISOTP (*dans le module socket*), 1168
CAN_J1939 (*dans le module socket*), 1168
CAN_RAW_FD_FRAMES (*dans le module socket*), 1168
CAN_RAW_JOIN_FILTERS (*dans le module socket*), 1168
can_symlink() (*dans le module test.support.os_helper*), 1860
can_write_eof() (*méthode asyncio.StreamWriter*), 1088
can_write_eof() (*méthode asyncio.WriteTransport*), 1137
can_xattr() (*dans le module test.support.os_helper*), 1860
CANCEL (*dans le module tkinter.messagebox*), 1638
cancel() (*méthode asyncio.Future*), 1133
cancel() (*méthode asyncio.Handle*), 1125
cancel() (*méthode asyncio.Task*), 1083
cancel() (*méthode concurrent.futures.Future*), 1025
cancel() (*méthode sched.scheduler*), 1048
cancel() (*méthode threading.Timer*), 967
cancel() (*méthode tkinter.dnd.DndHandler*), 1640
cancel_command() (*méthode tkinter.filedialog.FileDialog*), 1636
cancel_dump_traceback_later() (*dans le module faulthandler*), 1875
cancel_join_thread() (*méthode multiprocessing.Queue*), 981
cancelled() (*méthode asyncio.Future*), 1132
cancelled() (*méthode asyncio.Handle*), 1125
cancelled() (*méthode asyncio.Task*), 1083
cancelled() (*méthode concurrent.futures.Future*), 1026
CancelledError, 1027, 1105
cancelling() (*méthode asyncio.Task*), 1084
CannotSendHeader, 1459
CannotSendRequest, 1459
canonic() (*méthode bdb.Bdb*), 1870
canonical() (*méthode decimal.Context*), 366
canonical() (*méthode decimal.Decimal*), 357
canonicalize() (*dans le module xml.etree.ElementTree*), 1361
capa() (*méthode poplib.POP3*), 1472
capitalize() (*méthode bytearray*), 74
capitalize() (*méthode bytes*), 74
capitalize() (*méthode str*), 54
CapsuleType (*classe dans types*), 304
capture_call_graph() (*dans le module asyncio*), 1106
captured_stderr() (*dans le module test.support*), 1850
captured_stdin() (*dans le module test.support*), 1850
captured_stdout() (*dans le module test.support*), 1850
captureWarnings() (*dans le module logging*), 774
capwords() (*dans le module string*), 132
casefold() (*méthode str*), 54
cast() (*dans le module ctypes*), 839
cast() (*dans le module typing*), 1710
cast() (*méthode memoryview*), 84
--catch
    option de ligne de commande unittest, 1754
catch_threading_exception() (*dans le module test.support.threading_helper*), 1858
catch_unraisable_exception() (*dans le module test.support*), 1854
catch_warnings (*classe dans warnings*), 1978
category() (*dans le module unicodedata*), 172
cbreak() (*dans le module curses*), 924
cbrt() (*dans le module math*), 343
ccc() (*méthode ftplib.FTP_TLS*), 1470
C-contiguous, 2232
cdf() (*méthode statistics.NormalDist*), 405
CDLL (*classe dans ctypes*), 833
ceil() (*dans le module math*), 340

ceil() (*in module math*), 40
CellType (*dans le module types*), 302
center() (*méthode bytearray*), 71
center() (*méthode bytes*), 71
center() (*méthode str*), 54
CERT_NONE (*dans le module ssl*), 1195
CERT_OPTIONAL (*dans le module ssl*), 1195
CERT_REQUIRED (*dans le module ssl*), 1195
cert_store_stats() (*méthode ssl.SSLContext*), 1207
cert_time_to_seconds() (*dans le module ssl*), 1194
CertificateError, 1193
certificates, 1215
cfmakecbreak() (*dans le module tty*), 2201
cfmakeraw() (*dans le module tty*), 2201
CFUNCTYPE() (*dans le module ctypes*), 836
cget() (*méthode tkinter.font.Font*), 1633
cgi
    module, 2222
cgi_directories (*attribut http.server.CGIHTTPRequestHandler*), 1508
CGIHandler (*classe dans wsgiref.handlers*), 1421
CGIHTTPRequestHandler (*classe dans http.server*), 1507
cgitb
    module, 2222
CGIXMLRPCRequestHandler (*classe dans xmlrpc.server*), 1529
chain() (*dans le module itertools*), 412
chaîne de documentation (*docstring*), **2233**
chaîne entre triple guillemets, **2243**
chaining
    comparisons, 38
    exception, 107
ChainMap (*classe dans collections*), 259
ChainMap (*classe dans typing*), 1720
change_cwd() (*dans le module test.support.os_helper*), 1860
CHANNEL_BINDING_TYPES (*dans le module ssl*), 1200
CHAR_MAX (*dans le module locale*), 1568
character, 172
CharacterDataHandler() (*méthode xml.parsers.expat.xmlparser*), 1407
characters() (*méthode xml.sax.handler.ContentHandler*), 1396
characters_written (*attribut BlockingIOError*), 115
chargeur, **2238**
Charset (*classe dans email.charset*), 1301
charset() (*méthode gettext.NullTranslations*), 1556
chdir() (*dans le module contextlib*), 1994
chdir() (*dans le module os*), 687
check (*attribut lzma.LZMADecompressor*), 582
check() (*dans le module tabnanny*), 2152
check() (*méthode imaplib.IMAP4*), 1476
check__all__() (*dans le module test.support*), 1854
check_call() (*dans le module subprocess*), 1042
check_disallow_instantiation() (*dans le module test.support*), 1855
CHECK_EG_MATCH (*opcode*), 2170
CHECK_EXC_MATCH (*opcode*), 2170
check_free_after_iterating() (*dans le module test.support*), 1854
check_hostname (*attribut ssl.SSLContext*), 1212
check_impl_detail() (*dans le module test.support*), 1850
check_no_resource_warning() (*dans le module test.support.warnings_helper*), 1862
check_output() (*dans le module subprocess*), 1043
check_output() (*méthode doctest.OutputChecker*), 1747
check_returncode() (*méthode subprocess.CompletedProcess*), 1030
check_syntax_error() (*dans le module test.support*), 1853
check_syntax_warning() (*dans le module test.support.warnings_helper*), 1862
check_unused_args() (*méthode string.Formatter*), 123
check_warnings() (*dans le module test.support.warnings_helper*), 1863
checkcache() (*dans le module linecache*), 497
CHECKED_HASH (*attribut py_compile.PycInvalidationMode*), 2155
checkfuncname() (*dans le module bdb*), 1873
checksizeof() (*dans le module test.support*), 1852
checksum
    Cyclic Redundancy Check, 568
chemin des importations, **2236**
chercheur, **2234**
chercheur basé sur les chemins, **2241**
chercheur dans les méta-chemins, **2238**
chercheur de chemins, **2241**
chflags() (*dans le module os*), 687
chgat() (*méthode curses.window*), 932
childNodes (*attribut xml.dom.Node*), 1377
ChildProcessError, 115
children (*attribut pyclbr.Class*), 2153
children (*attribut pyclbr.Function*), 2153
children (*attribut tkinter.Tk*), 1621
chksum (*attribut tarfile.TarInfo*), 604
chmod() (*dans le module os*), 688
chmod() (*méthode pathlib.Path*), 469
--choice
    option de ligne de commande random, 392
choice() (*dans le module random*), 385
choice() (*dans le module secrets*), 659
choices (*attribut optparse.Option*), 908
choices() (*dans le module random*), 385
Chooser (*classe dans tkinter.colorchooser*), 1632
chown() (*dans le module os*), 688
chown() (*dans le module shutil*), 501
chr()
    built-in function, 10
chroot() (*dans le module os*), 689
CHRTYPE (*dans le module tarfile*), 598
chunk

module, 2222
cipher() (*méthode ssl.SSLSocket*), 1204
circle() (*dans le module turtle*), 1581
CIRCUMFLEX (*dans le module token*), 2145
CIRCUMFLEXEQUAL (*dans le module token*), 2145
Clamped (*classe dans decimal*), 370
CLASS (*attribut symtable.SymbolTableType*), 2139
Class (*classe dans pyclbr*), 2153
Class (*classe dans symtable*), 2141
ClassDef (*classe dans ast*), 2132
classe, **2231**
classe mère abstraite, **2229**
classmethod()
built-in function, 10
ClassMethodDescriptorType (*dans le module types*), 302
ClassVar (*dans le module typing*), 1688
CLD_CONTINUED (*dans le module os*), 724
CLD_DUMPED (*dans le module os*), 724
CLD_EXITED (*dans le module os*), 724
CLD_KILLED (*dans le module os*), 724
CLD_STOPPED (*dans le module os*), 724
CLD_TRAPPED (*dans le module os*), 724
clean() (*méthode mailbox.Maildir*), 1321
cleandoc() (*dans le module inspect*), 2031
CleanImport (*classe dans test.support.import_helper*), 1862
cleanup() (*méthode tempfile.TemporaryDirectory*), 489
CLEANUP_THROW (*opcode*), 2168
clear (*pdb command*), 1880
clear() (*dans le module turtle*), 1589
clear() (*méthode array.array*), 291
clear() (*méthode asyncio.Event*), 1094
clear() (*méthode collections.deque*), 265
clear() (*méthode curses.window*), 932
clear() (*méthode dbm.gnu.gdbm*), 534
clear() (*méthode dbm.ndbm.ndbm*), 535
clear() (*méthode dict*), 92
clear() (*méthode email.message.EmailMessage*), 1258
clear() (*méthode frozenset*), 90
clear() (*méthode http.cookiejar.CookieJar*), 1515
clear() (*méthode mailbox.Mailbox*), 1319
clear() (*méthode threading.Event*), 966
clear() (*méthode xml.etree.ElementTree.Element*), 1366
clear() (*sequence method*), 49
clear_all_breaks() (*méthode bdb.Bdb*), 1872
clear_all_file_breaks() (*méthode bdb.Bdb*), 1872
clear_bpbynumber() (*méthode bdb.Bdb*), 1872
clear_break() (*méthode bdb.Bdb*), 1872
clear_cache() (*dans le module filecmp*), 485
clear_cache() (*méthode de la classe zoneinfo.ZoneInfo*), 247
clear_content() (*méthode email.message.EmailMessage*), 1258
clear_flags() (*méthode decimal.Context*), 365
clear_frames() (*dans le module traceback*), 2013
clear_history() (*dans le module readline*), 177
clear_overloads() (*dans le module typing*), 1714
clear_session_cookies() (*méthode http.cookiejar.CookieJar*), 1515
clear_tool_id() (*dans le module sys.monitoring*), 1955
clear_traces() (*dans le module tracemalloc*), 1906
clear_traps() (*méthode decimal.Context*), 365
clearcache() (*dans le module linecache*), 496
clearok() (*méthode curses.window*), 932
clearscreen() (*dans le module turtle*), 1595
clearstamp() (*dans le module turtle*), 1582
clearstamps() (*dans le module turtle*), 1582
Client() (*dans le module multiprocessing.connection*), 1001
client_address (*attribut http.server.BaseHTTPRequestHandler*), 1503
client_address (*attribut socketserver.BaseRequestHandler*), 1498
CLOCK_BOOTTIME (*dans le module time*), 753
clock_getres() (*dans le module time*), 745
clock_gettime() (*dans le module time*), 745
clock_gettime_ns() (*dans le module time*), 745
CLOCK_HIGHRES (*dans le module time*), 753
CLOCK_MONOTONIC (*dans le module time*), 754
CLOCK_MONOTONIC_RAW (*dans le module time*), 754
CLOCK_MONOTONIC_RAW_APPROX (*dans le module time*), 754
CLOCK_PROCESS_CPUTIME_ID (*dans le module time*), 754
CLOCK_PROF (*dans le module time*), 754
CLOCK_REALTIME (*dans le module time*), 755
clock_seq (*attribut uuid.UUID*), 1490
clock_seq_hi_variant (*attribut uuid.UUID*), 1490
clock_seq_low (*attribut uuid.UUID*), 1490
clock_settime() (*dans le module time*), 745
clock_settime_ns() (*dans le module time*), 745
CLOCK_TAI (*dans le module time*), 754
CLOCK_THREAD_CPUTIME_ID (*dans le module time*), 754
CLOCK_UPTIME (*dans le module time*), 754
CLOCK_UPTIME_RAW (*dans le module time*), 754
CLOCK_UPTIME_RAW_APPROX (*dans le module time*), 755
clone() (*dans le module turtle*), 1594
clone() (*méthode email.generator.BytesGenerator*), 1263
clone() (*méthode email.generator.Generator*), 1264
clone() (*méthode email.policy.Policy*), 1267
CLONE_FILES (*dans le module os*), 672
CLONE_FS (*dans le module os*), 672
CLONE_NEWCGROUP (*dans le module os*), 672
CLONE_NEWIPC (*dans le module os*), 672
CLONE_NEWNET (*dans le module os*), 672
CLONE_NEWNS (*dans le module os*), 672
CLONE_NEWPID (*dans le module os*), 672

CLONE_NEWTIME (*dans le module os*), 672
CLONE_NEWUSER (*dans le module os*), 672
CLONE_NEWUTS (*dans le module os*), 672
CLONE_SIGHAND (*dans le module os*), 672
CLONE_SYSVSEM (*dans le module os*), 672
CLONE_THREAD (*dans le module os*), 672
CLONE_VM (*dans le module os*), 672
cloneNode() (*méthode xml.dom.Node*), 1378
close() (*dans le module fileinput*), 922
close() (*dans le module os*), 673
close() (*dans le module socket*), 1173
close() (*méthode asyncio.BaseTransport*), 1136
close() (*méthode asyncio.loop*), 1109
close() (*méthode asyncio.Runner*), 1064
close() (*méthode asyncio.Server*), 1126
close() (*méthode asyncio.StreamWriter*), 1088
close() (*méthode asyncio.SubprocessTransport*), 1138
close() (*méthode contextlib.ExitStack*), 1997
close() (*méthode dbm.dumb.dumbdbm*), 536
close() (*méthode dbm.gnu.gdbm*), 534
close() (*méthode dbm.ndbm.ndbm*), 535
close() (*méthode email.parser.BytesFeedParser*), 1259
close() (*méthode ftplib.FTP*), 1469
close() (*méthode html.parser.HTMLParser*), 1349
close() (*méthode http.client.HTTPConnection*), 1461
close() (*méthode imaplib.IMAP4*), 1476
close() (*méthode io.IOBase*), 734
close() (*méthode logging.FileHandler*), 787
close() (*méthode logging.Handler*), 763
close() (*méthode logging.handlers.MemoryHandler*), 797
close() (*méthode logging.handlers.NTEventLogHandler*), 795
close() (*méthode logging.handlers.SocketHandler*), 791
close() (*méthode logging.handlers.SysLogHandler*), 793
close() (*méthode mailbox.Mailbox*), 1320
close() (*méthode mailbox.Maildir*), 1323
close() (*méthode mailbox.MH*), 1325
close() (*méthode mmap.mmap*), 1246
close() (*méthode multiprocessing.connection.Connection*), 985
close() (*méthode multiprocessing.connection.Listener*), 1001
close() (*méthode multiprocessing.pool.Pool*), 999
close() (*méthode multiprocessing.Process*), 978
close() (*méthode multiprocessing.Queue*), 981
close() (*méthode multiprocessing.shared_memory.SharedMemory*), 1015
close() (*méthode multiprocessing.SimpleQueue*), 982
close() (*méthode os.scandir*), 695
close() (*méthode select.devpoll*), 1226
close() (*méthode select.epoll*), 1227
close() (*méthode select.kqueue*), 1229
close() (*méthode selectors.BaseSelector*), 1233
close() (*méthode shelve.Shelf*), 527
close() (*méthode socket.socket*), 1179
close() (*méthode sqlite3.Blob*), 555
close() (*méthode sqlite3.Connection*), 543
close() (*méthode sqlite3.Cursor*), 554
close() (*méthode tarfile.TarFile*), 602
close() (*méthode urllib.request.BaseHandler*), 1434
close() (*méthode wave.Wave_read*), 1550
close() (*méthode wave.Wave_write*), 1551
Close() (*méthode winreg.PyHKEY*), 2194
close() (*méthode xml.etree.ElementTree.TreeBuilder*), 1370
close() (*méthode xml.etree.ElementTree.XMLParser*), 1371
close() (*méthode xml.etree.ElementTree.XMLPullParser*), 1373
close() (*méthode xml.sax.xmlreader.IncrementalParser*), 1401
close() (*méthode zipfile.ZipFile*), 587
close_clients() (*méthode asyncio.Server*), 1126
close_connection (*attribut http.server.BaseHTTPRequestHandler*), 1503
closed (*attribut http.client.HTTPResponse*), 1463
closed (*attribut io.IOBase*), 734
closed (*attribut mmap.mmap*), 1246
closed (*attribut select.devpoll*), 1226
closed (*attribut select.epoll*), 1227
closed (*attribut select.kqueue*), 1229
CloseKey() (*dans le module winreg*), 2186
closelog() (*dans le module syslog*), 2211
closerange() (*dans le module os*), 673
closing() (*dans le module contextlib*), 1991
closure variable, **2231**
clrtobot() (*méthode curses.window*), 932
clrtoeol() (*méthode curses.window*), 932
cmath
    module, 347
cmd
    module, 1608, 1876
cmd (*attribut subprocess.CalledProcessError*), 1031
cmd (*attribut subprocess.TimeoutExpired*), 1030
Cmd (*classe dans cmd*), 1608
cmdloop() (*méthode cmd.Cmd*), 1608
cmdqueue (*attribut cmd.Cmd*), 1610
cmp() (*dans le module filecmp*), 485
cmp_op (*dans le module dis*), 2180
cmp_to_key() (*dans le module functools*), 428
cmpfiles() (*dans le module filecmp*), 485
CMSG_LEN() (*dans le module socket*), 1176
CMSG_SPACE() (*dans le module socket*), 1177
CO_ASYNC_GENERATOR (*dans le module inspect*), 2043
CO_COROUTINE (*dans le module inspect*), 2043
CO_GENERATOR (*dans le module inspect*), 2043
CO_HAS_DOCSTRING (*dans le module inspect*), 2043
CO_ITERABLE_COROUTINE (*dans le module inspect*), 2043
CO_METHOD (*dans le module inspect*), 2043
CO_NESTED (*dans le module inspect*), 2043
CO_NEWLOCALS (*dans le module inspect*), 2043

CO_OPTIMIZED (*dans le module inspect*), 2043
CO_VARARGS (*dans le module inspect*), 2043
CO_VARKEYWORDS (*dans le module inspect*), 2043
code
    module, 2053
code (*attribut SystemExit*), 113
code (*attribut urllib.error.HTTPError*), 1452
code (*attribut urllib.response.addinfourl*), 1442
code (*attribut xml.etree.ElementTree.ParseError*), 1373
code (*attribut xml.parsers.expat.ExpatError*), 1408
code intermédiaire (*bytecode*), **2231**
code object, 103, 529
code_context (*attribut inspect.FrameInfo*), 2039
code_context (*attribut inspect.Traceback*), 2039
code_info() (*dans le module dis*), 2162
Codec (*classe dans codecs*), 194
CodecInfo (*classe dans codecs*), 189
Codecs, 189
    decode, 189
    encode, 189
codecs
    module, 189
coded_value (*attribut http.cookies.Morsel*), 1511
codeop
    module, 2055
codepoint2name (*dans le module html.entities*), 1352
codes (*dans le module xml.parsers.expat.errors*), 1410
CODESET (*dans le module locale*), 1563
CodeType (*classe dans types*), 302
col_offset (*attribut ast.AST*), 2105
collapse_addresses() (*dans le module ipaddress*), 1547
collapse_rfc2231_value() (*dans le module email.utils*), 1306
collect() (*dans le module gc*), 2022
collectedDurations (*attribut unittest.TestResult*), 1777
Collection (*classe dans collections.abc*), 279
Collection (*classe dans typing*), 1721
collections
    module, 258
collections.abc
    module, 276
colno (*attribut json.JSONDecodeError*), 1314
colno (*attribut re.PatternError*), 146
colno (*attribut tomllib.TOMLDecodeError*), 640
colon (*attribut mailbox.Maildir*), 1321
COLON (*dans le module token*), 2144
COLONEQUAL (*dans le module token*), 2146
color() (*dans le module turtle*), 1587
COLOR_BLACK (*dans le module curses*), 948
COLOR_BLUE (*dans le module curses*), 948
color_content() (*dans le module curses*), 924
COLOR_CYAN (*dans le module curses*), 948
COLOR_GREEN (*dans le module curses*), 948
COLOR_MAGENTA (*dans le module curses*), 948
color_pair() (*dans le module curses*), 924
COLOR_PAIRS (*dans le module curses*), 937
COLOR_RED (*dans le module curses*), 948
COLOR_WHITE (*dans le module curses*), 948
COLOR_YELLOW (*dans le module curses*), 948
colormode() (*dans le module turtle*), 1600
COLORS (*dans le module curses*), 937
colorsys
    module, 1552
COLS (*dans le module curses*), 937
column() (*méthode tkinter.ttk.Treeview*), 1651
columnize() (*méthode cmd.Cmd*), 1609
COLUMNS, 930
columns (*attribut os.terminal_size*), 685
comb() (*dans le module math*), 340
combinations() (*dans le module itertools*), 412
combinations_with_replacement() (*dans le module itertools*), 413
combine() (*méthode de la classe datetime.datetime*), 220
combining() (*dans le module unicodedata*), 172
Combobox (*classe dans tkinter.ttk*), 1645
COMError, 848
COMMA (*dans le module token*), 2144
command (*attribut http.server.BaseHTTPRequestHandler*), 1503
CommandCompiler (*classe dans codeop*), 2056
commands (*pdb command*), 1881
comment (*attribut http.cookiejar.Cookie*), 1519
comment (*attribut http.cookies.Morsel*), 1510
comment (*attribut zipfile.ZipFile*), 590
comment (*attribut zipfile.ZipInfo*), 593
COMMENT (*dans le module token*), 2146
Comment() (*dans le module xml.etree.ElementTree*), 1362
comment() (*méthode xml.etree.ElementTree.TreeBuilder*), 1371
comment() (*méthode xml.sax.handler.LexicalHandler*), 1397
comment_url (*attribut http.cookiejar.Cookie*), 1519
commenters (*attribut shlex.shlex*), 1615
CommentHandler() (*méthode xml.parsers.expat.xmlparser*), 1407
commit() (*méthode sqlite3.Connection*), 543
common (*attribut filecmp.dircmp*), 486
Common Vulnerabilities and Exposures
    CVE 2020-10735, 104
    CVE 2023-52425, 1354
Common Weakness Enumeration
    CWE 257, 660
common_dirs (*attribut filecmp.dircmp*), 486
common_files (*attribut filecmp.dircmp*), 486
common_funny (*attribut filecmp.dircmp*), 486
common_types (*dans le module mimetypes*), 1337
commonpath() (*dans le module os.path*), 473
commonprefix() (*dans le module os.path*), 473
communicate() (*méthode asyncio.subprocess.Process*), 1100
communicate() (*méthode subprocess.Popen*), 1038
--compact

option de ligne de commande json, 1317
Compare (*classe dans ast*), 2111
compare() (*dans le module ast*), 2138
compare() (*méthode decimal.Context*), 366
compare() (*méthode decimal.Decimal*), 357
compare() (*méthode difflib.Differ*), 164
compare_digest() (*dans le module hmac*), 658
compare_digest() (*dans le module secrets*), 660
compare_networks() (*méthode ipaddress.IPv4Network*), 1543
compare_networks() (*méthode ipaddress.IPv6Network*), 1544
COMPARE_OP (*opcode*), 2173
compare_signal() (*méthode decimal.Context*), 366
compare_signal() (*méthode decimal.Decimal*), 357
compare_to() (*méthode tracemalloc.Snapshot*), 1908
compare_total() (*méthode decimal.Context*), 366
compare_total() (*méthode decimal.Decimal*), 357
compare_total_mag() (*méthode decimal.Context*), 366
compare_total_mag() (*méthode decimal.Decimal*), 358
comparing
objects, 38
comparison
operator, 38
COMPARISON_FLAGS (*dans le module doctest*), 1736
comparisons
chaining, 38
Compat32 (*classe dans email.policy*), 1271
compat32 (*dans le module email.policy*), 1271
compile
built-in function, 103, 302
Compile (*classe dans codeop*), 2056
compile()
built-in function, 11
compile() (*dans le module py_compile*), 2154
compile() (*dans le module re*), 142
compile_command() (*dans le module code*), 2054
compile_command() (*dans le module codeop*), 2055
compile_dir() (*dans le module compileall*), 2157
compile_file() (*dans le module compileall*), 2158
compile_path() (*dans le module compileall*), 2159
compileall
module, 2156
compiler_flag (*attribut __future__._Feature*), 2021
complete() (*méthode rlcompleter.Completer*), 180
complete_statement() (*dans le module sqlite3*), 540
completedefault() (*méthode cmd.Cmd*), 1609
CompletedProcess (*classe dans subprocess*), 1030
Completer (*classe dans rlcompleter*), 180
complex
built-in function, 39
Complex (*classe dans numbers*), 335
complex (*classe de base*), 12
complex number
literals, 39
object, 38
comprehension (*classe dans ast*), 2114
--compress
option de ligne de commande zipapp, 1925
compress() (*dans le module bz2*), 577
compress() (*dans le module gzip*), 573
compress() (*dans le module itertools*), 413
compress() (*dans le module lzma*), 582
compress() (*dans le module zlib*), 567
compress() (*méthode bz2.BZ2Compressor*), 576
compress() (*méthode lzma.LZMACompressor*), 581
compress() (*méthode zlib.Compress*), 569
compress_size (*attribut zipfile.ZipInfo*), 594
compress_type (*attribut zipfile.ZipInfo*), 593
compressed (*attribut ipaddress.IPv4Address*), 1536
compressed (*attribut ipaddress.IPv4Network*), 1541
compressed (*attribut ipaddress.IPv6Address*), 1538
compressed (*attribut ipaddress.IPv6Network*), 1544
compression() (*méthode ssl.SSLSocket*), 1204
CompressionError, 597
compressobj() (*dans le module zlib*), 568
COMSPEC, 721, 1033
concat() (*dans le module operator*), 439
Concatenate (*dans le module typing*), 1687
concatenation
operation, 47
concurrent.futures
module, 1020
cond (*attribut bdb.Breakpoint*), 1870
Condition (*classe dans asyncio*), 1094
Condition (*classe dans multiprocessing*), 986
Condition (*classe dans threading*), 964
condition (*pdb command*), 1881
Condition() (*méthode multiprocessing.managers.SyncManager*), 993
config() (*méthode tkinter.font.Font*), 1633
configparser
module, 620
ConfigParser (*classe dans configparser*), 633
configuration
file, 620
file, debugger, 1880
file, path, 2050
configuration information, 1959
configure() (*méthode tkinter.ttk.Style*), 1655
configure_mock() (*méthode unittest.mock.Mock*), 1789
CONFORM (*attribut enum.FlagBoundary*), 327
confstr() (*dans le module os*), 727
confstr_names (*dans le module os*), 727
conjugate() (*complex number method*), 39
conjugate() (*méthode decimal.Decimal*), 358
conjugate() (*méthode numbers.Complex*), 335
connect() (*dans le module sqlite3*), 539
connect() (*méthode ftplib.FTP*), 1466
connect() (*méthode http.client.HTTPConnection*), 1461

connect() (*méthode multiprocessing.managers.BaseManager*), 992
connect() (*méthode smtplib.SMTP*), 1484
connect() (*méthode socket.socket*), 1179
connect_accepted_socket() (*méthode asyncio.loop*), 1116
connect_ex() (*méthode socket.socket*), 1179
connect_read_pipe() (*méthode asyncio.loop*), 1120
connect_write_pipe() (*méthode asyncio.loop*), 1121
connection (*attribut sqlite3.Cursor*), 554
Connection (*classe dans multiprocessing.connection*), 984
Connection (*classe dans sqlite3*), 542
connection_lost() (*méthode asyncio.BaseProtocol*), 1139
connection_made() (*méthode asyncio.BaseProtocol*), 1139
ConnectionAbortedError, 115
ConnectionError, 115
ConnectionRefusedError, 115
ConnectionResetError, 115
ConnectRegistry() (*dans le module winreg*), 2186
const (*attribut optparse.Option*), 908
Constant (*classe dans ast*), 2107
constructor() (*dans le module copyreg*), 525
consumed (*attribut asyncio.LimitOverrunError*), 1105
container
    iteration over, 46
Container (*classe dans collections.abc*), 279
Container (*classe dans typing*), 1721
contains() (*dans le module operator*), 439
CONTAINS_OP (*opcode*), 2173
content (*attribut urllib.error.ContentTooShortError*), 1452
content type
    MIME, 1336
content_disposition (*attribut email.headerregistry.ContentDispositionHeader*), 1276
content_manager (*attribut email.policy.EmailPolicy*), 1269
content_type (*attribut email.headerregistry.ContentTypeHeader*), 1276
ContentDispositionHeader (*classe dans email.headerregistry*), 1276
ContentHandler (*classe dans xml.sax.handler*), 1393
ContentManager (*classe dans email.contentmanager*), 1279
contents (*attribut ctypes._Pointer*), 848
contents() (*dans le module importlib.resources*), 2087
contents() (*méthode importlib.abc.ResourceReader*), 2073
contents() (*méthode importlib.resources.abc.ResourceReader*), 2088
ContentTooShortError, 1452
ContentTransferEncoding (*classe dans email.headerregistry*), 1276
ContentTypeHeader (*classe dans email.headerregistry*), 1276
context, **2232**
context (*attribut ssl.SSLSocket*), 1205
Context (*classe dans contextvars*), 1054
Context (*classe dans decimal*), 364
context management protocol, 95, **2232**
context manager, 95
context_diff() (*dans le module difflib*), 157
ContextDecorator (*classe dans contextlib*), 1994
contextlib
    module, 1989
ContextManager (*classe dans typing*), 1723
contextmanager() (*dans le module contextlib*), 1989
ContextVar (*classe dans contextvars*), 1053
contextvars
    module, 1052
CONTIG (*attribut inspect.BufferFlags*), 2044
CONTIG_RO (*attribut inspect.BufferFlags*), 2044
contigu, **2232**
contiguous (*attribut memoryview*), 88
Continue (*classe dans ast*), 2120
continue (*pdb command*), 1881
CONTINUOUS (*attribut enum.EnumCheck*), 326
control() (*méthode select.kqueue*), 1229
controlnames (*dans le module curses.ascii*), 953
CONTTYPE (*dans le module tarfile*), 598
conversions
    numeric, 40
convert_arg_line_to_args() (*méthode argparse.ArgumentParser*), 878
convert_field() (*méthode string.Formatter*), 123
CONVERT_VALUE (*opcode*), 2177
Cookie (*classe dans http.cookiejar*), 1513
CookieError, 1509
cookiejar (*attribut urllib.request.HTTPCookieProcessor*), 1436
CookieJar (*classe dans http.cookiejar*), 1513
CookiePolicy (*classe dans http.cookiejar*), 1513
Coordinated Universal Time, 744
copy
    module, 306, 525
    protocol, 516
COPY (*opcode*), 2166
copy() (*dans le module copy*), 306
copy() (*dans le module multiprocessing.sharedctypes*), 990
copy() (*dans le module shutil*), 498
copy() (*méthode collections.deque*), 265
copy() (*méthode contextvars.Context*), 1055
copy() (*méthode decimal.Context*), 365
copy() (*méthode dict*), 92
copy() (*méthode frozenset*), 89
copy() (*méthode hashlib.hash*), 648
copy() (*méthode hmac.HMAC*), 658
copy() (*méthode http.cookies.Morsel*), 1511
copy() (*méthode imaplib.IMAP4*), 1476

copy() (*méthode pathlib.Path*), 467
copy() (*méthode tkinter.font.Font*), 1633
copy() (*méthode types.MappingProxyType*), 304
copy() (*méthode zlib.Compress*), 569
copy() (*méthode zlib.Decompress*), 570
copy() (*sequence method*), 49
copy2() (*dans le module shutil*), 499
copy_abs() (*méthode decimal.Context*), 366
copy_abs() (*méthode decimal.Decimal*), 358
copy_context() (*dans le module contextvars*), 1054
copy_decimal() (*méthode decimal.Context*), 365
copy_file_range() (*dans le module os*), 673
COPY_FREE_VARS (*opcode*), 2176
copy_into() (*méthode pathlib.Path*), 467
copy_location() (*dans le module ast*), 2136
copy_negate() (*méthode decimal.Context*), 366
copy_negate() (*méthode decimal.Decimal*), 358
copy_sign() (*méthode decimal.Context*), 366
copy_sign() (*méthode decimal.Decimal*), 358
CopyComPointer() (*dans le module ctypes*), 839
copyfile() (*dans le module shutil*), 497
copyfileobj() (*dans le module shutil*), 497
copying files, 497
copymode() (*dans le module shutil*), 498
copyreg
    module, 525
copyright (*dans le module sys*), 1931
copyright (*variable de base*), 36
copysign() (*dans le module math*), 341
copystat() (*dans le module shutil*), 498
copytree() (*dans le module shutil*), 499
coroutine, **2232**
Coroutine (*classe dans collections.abc*), 280
Coroutine (*classe dans typing*), 1722
coroutine() (*dans le module types*), 305
CoroutineType (*dans le module types*), 301
correlation() (*dans le module statistics*), 403
cos() (*dans le module cmath*), 349
cos() (*dans le module math*), 345
cosh() (*dans le module cmath*), 349
cosh() (*dans le module math*), 346
--count
    option de ligne de commande trace, 1899
count (*attribut tracemalloc.Statistic*), 1909
count (*attribut tracemalloc.StatisticDiff*), 1909
count() (*dans le module itertools*), 414
count() (*méthode array.array*), 290
count() (*méthode bytearray*), 68
count() (*méthode bytes*), 68
count() (*méthode collections.deque*), 265
count() (*méthode memoryview*), 86
count() (*méthode multiprocessing.shared_memory.ShareableList*), 1018
count() (*méthode str*), 55
count() (*sequence method*), 47
count_diff (*attribut tracemalloc.StatisticDiff*), 1909
Counter (*classe dans collections*), 261
Counter (*classe dans typing*), 1720
countOf() (*dans le module operator*), 440
countTestCases() (*méthode unittest.TestCase*), 1770
countTestCases() (*méthode unittest.TestSuite*), 1773
covariance() (*dans le module statistics*), 402
CoverageResults (*classe dans trace*), 1900
--coverdir
    option de ligne de commande trace, 1899
cProfile
    module, 1888
CPU time, 747, 752
cpu_count() (*dans le module multiprocessing*), 982
cpu_count() (*dans le module os*), 727
CPython, **2232**
cpython_only() (*dans le module test.support*), 1853
CR (*dans le module curses.ascii*), 950
crawl_delay() (*méthode urllib.robotparser.RobotFileParser*), 1453
CRC (*attribut zipfile.ZipInfo*), 594
crc32() (*dans le module binascii*), 1344
crc32() (*dans le module zlib*), 568
crc_hqx() (*dans le module binascii*), 1344
--create
    option de ligne de commande tarfile, 609
    option de ligne de commande zipfile, 595
create() (*dans le module venv*), 1920
create() (*méthode imaplib.IMAP4*), 1476
create() (*méthode venv.EnvBuilder*), 1918
create_aggregate() (*méthode sqlite3.Connection*), 544
create_archive() (*dans le module zipapp*), 1925
create_autospec() (*dans le module unittest.mock*), 1817
CREATE_BREAKAWAY_FROM_JOB (*dans le module subprocess*), 1042
create_collation() (*méthode sqlite3.Connection*), 546
create_configuration() (*méthode venv.EnvBuilder*), 1919
create_connection() (*dans le module socket*), 1172
create_connection() (*méthode asyncio.loop*), 1112
create_datagram_endpoint() (*méthode asyncio.loop*), 1114
create_decimal() (*méthode decimal.Context*), 365
create_decimal_from_float() (*méthode decimal.Context*), 366
create_default_context() (*dans le module ssl*), 1191
CREATE_DEFAULT_ERROR_MODE (*dans le module subprocess*), 1042
create_eager_task_factory() (*dans le module asyncio*), 1073
create_empty_file() (*dans le module test.support.os_helper*), 1860
create_function() (*méthode sqlite3.Connection*),

543
create_future() (*méthode asyncio.loop*), 1112
create_git_ignore_file() (*méthode venv.EnvBuilder*), 1920
create_module() (*méthode importlib.abc.Loader*), 2069
create_module() (*méthode importlib.machinery.ExtensionFileLoader*), 2077
create_module() (*méthode zipimport.zipimporter*), 2058
CREATE_NEW_CONSOLE (*dans le module subprocess*), 1041
CREATE_NEW_PROCESS_GROUP (*dans le module subprocess*), 1041
CREATE_NO_WINDOW (*dans le module subprocess*), 1041
create_server() (*dans le module socket*), 1172
create_server() (*méthode asyncio.loop*), 1115
create_stats() (*méthode profile.Profile*), 1889
create_string_buffer() (*dans le module ctypes*), 839
create_subprocess_exec() (*dans le module asyncio*), 1098
create_subprocess_shell() (*dans le module asyncio*), 1098
create_system (*attribut zipfile.ZipInfo*), 593
create_task() (*dans le module asyncio*), 1068
create_task() (*méthode asyncio.loop*), 1112
create_task() (*méthode asyncio.TaskGroup*), 1069
create_unicode_buffer() (*dans le module ctypes*), 839
create_unix_connection() (*méthode asyncio.loop*), 1115
create_unix_server() (*méthode asyncio.loop*), 1116
create_version (*attribut zipfile.ZipInfo*), 594
create_window_function() (*méthode sqlite3.Connection*), 544
createAttribute() (*méthode xml.dom.Document*), 1379
createAttributeNS() (*méthode xml.dom.Document*), 1379
createComment() (*méthode xml.dom.Document*), 1379
createDocument() (*méthode xml.dom.DOMImplementation*), 1376
createDocumentType() (*méthode xml.dom.DOMImplementation*), 1376
createElement() (*méthode xml.dom.Document*), 1379
createElementNS() (*méthode xml.dom.Document*), 1379
createfilehandler() (*méthode _tkinter.Widget.tk*), 1632
CreateKey() (*dans le module winreg*), 2186
CreateKeyEx() (*dans le module winreg*), 2186
createLock() (*méthode logging.Handler*), 762
createLock() (*méthode logging.NullHandler*), 787
createProcessingInstruction() (*méthode xml.dom.Document*), 1379
createSocket() (*méthode logging.handlers.SocketHandler*), 792
createSocket() (*méthode logging.handlers.SysLogHandler*), 793
createTextNode() (*méthode xml.dom.Document*), 1379
credits (*variable de base*), 36
CRITICAL (*dans le module logging*), 762
critical() (*dans le module logging*), 770
critical() (*méthode logging.Logger*), 761
CRNCYSTR (*dans le module locale*), 1564
CRT_ASSEMBLY_VERSION (*dans le module msvcrt*), 2185
CRT_ASSERT (*dans le module msvcrt*), 2185
CRT_ERROR (*dans le module msvcrt*), 2185
CRT_WARN (*dans le module msvcrt*), 2185
CRTDBG_MODE_DEBUG (*dans le module msvcrt*), 2185
CRTDBG_MODE_FILE (*dans le module msvcrt*), 2185
CRTDBG_MODE_WNDW (*dans le module msvcrt*), 2185
CRTDBG_REPORT_MODE (*dans le module msvcrt*), 2185
CrtSetReportFile() (*dans le module msvcrt*), 2185
CrtSetReportMode() (*dans le module msvcrt*), 2185
crypt
    module, 2222
cryptography, 645
--css
    option de ligne de commande calendar, 258
cssclass_month (*attribut calendar.HTMLCalendar*), 253
cssclass_month_head (*attribut calendar.HTMLCalendar*), 253
cssclass_noday (*attribut calendar.HTMLCalendar*), 253
cssclass_year (*attribut calendar.HTMLCalendar*), 253
cssclass_year_head (*attribut calendar.HTMLCalendar*), 253
cssclasses (*attribut calendar.HTMLCalendar*), 253
cssclasses_weekday_head (*attribut calendar.HTMLCalendar*), 253
csv, 613
    module, 613
cte (*attribut email.headerregistry.ContentTransferEncoding*), 1276
cte_type (*attribut email.policy.Policy*), 1266
ctermid() (*dans le module os*), 665
ctime() (*dans le module time*), 745
ctime() (*méthode datetime.date*), 216
ctime() (*méthode datetime.datetime*), 226
ctrl() (*dans le module curses.ascii*), 952
CTRL_BREAK_EVENT (*dans le module signal*), 1237
CTRL_C_EVENT (*dans le module signal*), 1237
ctypes
    module, 812
curdir (*dans le module os*), 728
currency() (*dans le module locale*), 1566

current context, **2232**
current() (*méthode tkinter.ttk.Combobox*), 1645
current_process() (*dans le module multiprocessing*), 983
current_task() (*dans le module asyncio*), 1080
current_thread() (*dans le module threading*), 956
CurrentByteIndex (*attribut xml.parsers.expat.xmlparser*), 1406
CurrentColumnNumber (*attribut xml.parsers.expat.xmlparser*), 1406
currentframe() (*dans le module inspect*), 2040
CurrentLineNumber (*attribut xml.parsers.expat.xmlparser*), 1406
curs_set() (*dans le module curses*), 925
curses
    module, 923
curses.ascii
    module, 949
curses.panel
    module, 953
curses.textpad
    module, 948
Cursor (*classe dans sqlite3*), 552
cursor() (*méthode sqlite3.Connection*), 543
cursyncup() (*méthode curses.window*), 932
cwd() (*méthode de la classe pathlib.Path*), 459
cwd() (*méthode ftplib.FTP*), 1469
cycle() (*dans le module itertools*), 414
CycleError, 333
Cyclic Redundancy Check, 568

D

-d
    option de ligne de commande compileall, 2156
    option de ligne de commande gzip, 574
D_FMT (*dans le module locale*), 1563
D_T_FMT (*dans le module locale*), 1563
daemon (*attribut multiprocessing.Process*), 977
daemon (*attribut threading.Thread*), 961
daemon_threads (*attribut socketserver.ThreadingMixIn*), 1495
data
    packing binary, 181
    tabular, 613
data (*attribut collections.UserDict*), 275
data (*attribut collections.UserList*), 276
data (*attribut collections.UserString*), 276
data (*attribut select.kevent*), 1230
data (*attribut selectors.SelectorKey*), 1232
data (*attribut urllib.request.Request*), 1432
data (*attribut xml.dom.Comment*), 1381
data (*attribut xml.dom.ProcessingInstruction*), 1382
data (*attribut xml.dom.Text*), 1381
data (*attribut xmlrpc.client.Binary*), 1524
data() (*méthode xml.etree.ElementTree.TreeBuilder*), 1370
data_filter() (*dans le module tarfile*), 606
data_open() (*méthode urllib.request.DataHandler*), 1438
data_received() (*méthode asyncio.Protocol*), 1140
database
    Unicode, 172
DatabaseError, 556
databases, 535
dataclass() (*dans le module dataclasses*), 1979
dataclass_transform() (*dans le module typing*), 1711
dataclasses
    module, 1978
DataError, 556
datagram_received() (*méthode asyncio.DatagramProtocol*), 1141
DatagramHandler (*classe dans logging.handlers*), 792
DatagramProtocol (*classe dans asyncio*), 1139
DatagramRequestHandler (*classe dans socketserver*), 1498
DatagramTransport (*classe dans asyncio*), 1135
DataHandler (*classe dans urllib.request*), 1431
date (*classe dans datetime*), 213
date() (*méthode datetime.datetime*), 223
date_time (*attribut zipfile.ZipInfo*), 593
date_time_string() (*méthode http.server.BaseHTTPRequestHandler*), 1506
DateHeader (*classe dans email.headerregistry*), 1274
datetime
    module, 207
datetime (*attribut email.headerregistry.DateHeader*), 1275
datetime (*classe dans datetime*), 218
DateTime (*classe dans xmlrpc.client*), 1524
day (*attribut datetime.date*), 214
day (*attribut datetime.datetime*), 222
Day (*classe dans calendar*), 255
DAY_1 (*dans le module locale*), 1563
DAY_2 (*dans le module locale*), 1563
DAY_3 (*dans le module locale*), 1563
DAY_4 (*dans le module locale*), 1563
DAY_5 (*dans le module locale*), 1563
DAY_6 (*dans le module locale*), 1563
DAY_7 (*dans le module locale*), 1563
day_abbr (*dans le module calendar*), 255
day_name (*dans le module calendar*), 255
daylight (*dans le module time*), 755
Daylight Saving Time, 744
days (*attribut datetime.timedelta*), 210
DbfilenameShelf (*classe dans shelve*), 528
dbm
    module, 531
dbm.dumb
    module, 535
dbm.gnu
    module, 527, 533
dbm.ndbm
    module, 527, 534

dbm.sqlite3
module, 532
DC1 (*dans le module curses.ascii*), 951
DC2 (*dans le module curses.ascii*), 951
DC3 (*dans le module curses.ascii*), 951
DC4 (*dans le module curses.ascii*), 951
dcgettext() (*dans le module locale*), 1569
deactivate_stack_trampoline() (*dans le module sys*), 1951
debug (*attribut imaplib.IMAP4*), 1481
debug (*attribut shlex.shlex*), 1616
debug (*attribut sys.flags*), 1936
debug (*attribut zipfile.ZipFile*), 590
DEBUG (*dans le module logging*), 762
DEBUG (*dans le module re*), 140
debug (*pdb command*), 1884
debug() (*dans le module doctest*), 1749
debug() (*dans le module logging*), 770
debug() (*méthode logging.Logger*), 759
debug() (*méthode unittest.TestCase*), 1762
debug() (*méthode unittest.TestSuite*), 1773
DEBUG_BYTECODE_SUFFIXES (*dans le module importlib.machinery*), 2074
DEBUG_COLLECTABLE (*dans le module gc*), 2025
DEBUG_LEAK (*dans le module gc*), 2025
DEBUG_SAVEALL (*dans le module gc*), 2025
debug_src() (*dans le module doctest*), 1749
DEBUG_STATS (*dans le module gc*), 2025
DEBUG_UNCOLLECTABLE (*dans le module gc*), 2025
debugger, 957, 1662, 1941, 1949
configuration file, 1880
debugging, 1876
debuglevel (*attribut http.client.HTTPResponse*), 1463
DebugRunner (*classe dans doctest*), 1750
DECEMBER (*dans le module calendar*), 256
decimal
module, 351
Decimal (*classe dans decimal*), 355
decimal() (*dans le module unicodedata*), 172
DecimalException (*classe dans decimal*), 370
decode
Codecs, 189
decode (*attribut codecs.CodecInfo*), 189
decode() (*dans le module base64*), 1342
decode() (*dans le module codecs*), 189
decode() (*dans le module quopri*), 1345
decode() (*méthode bytearray*), 69
decode() (*méthode bytes*), 69
decode() (*méthode codecs.Codec*), 194
decode() (*méthode codecs.IncrementalDecoder*), 195
decode() (*méthode json.JSONDecoder*), 1312
decode() (*méthode xmlrpc.client.Binary*), 1524
decode() (*méthode xmlrpc.client.DateTime*), 1524
decode_header() (*dans le module email.header*), 1300
decode_params() (*dans le module email.utils*), 1306
decode_rfc2231() (*dans le module email.utils*), 1306
decode_source() (*dans le module importlib.util*), 2080
decodebytes() (*dans le module base64*), 1342
DecodedGenerator (*classe dans email.generator*), 1264
decodestring() (*dans le module quopri*), 1345
decomposition() (*dans le module unicodedata*), 173
--decompress
option de ligne de commande gzip, 574
decompress() (*dans le module bz2*), 577
decompress() (*dans le module gzip*), 573
decompress() (*dans le module lzma*), 582
decompress() (*dans le module zlib*), 568
decompress() (*méthode bz2.BZ2Decompressor*), 577
decompress() (*méthode lzma.LZMADecompressor*), 582
decompress() (*méthode zlib.Decompress*), 570
decompressobj() (*dans le module zlib*), 569
décorateur, **2233**
DEDENT (*dans le module token*), 2144
dedent() (*dans le module textwrap*), 169
deepcopy() (*dans le module copy*), 306
def_prog_mode() (*dans le module curses*), 925
def_shell_mode() (*dans le module curses*), 925
default (*attribut inspect.Parameter*), 2034
default (*attribut optparse.Option*), 908
default (*dans le module email.policy*), 1270
DEFAULT (*dans le module unittest.mock*), 1816
default() (*méthode cmd.Cmd*), 1609
default() (*méthode json.JSONEncoder*), 1313
DEFAULT_BUFFER_SIZE (*dans le module io*), 732
default_bufsize (*dans le module xml.dom.pulldom*), 1390
default_exception_handler() (*méthode asyncio.loop*), 1123
default_factory (*attribut collections.defaultdict*), 268
DEFAULT_FORMAT (*dans le module tarfile*), 599
DEFAULT_IGNORES (*dans le module filecmp*), 486
default_loader() (*dans le module xml.etree.ElementInclude*), 1366
default_max_str_digits (*attribut sys.int_info*), 1944
default_open() (*méthode urllib.request.BaseHandler*), 1434
DEFAULT_PROTOCOL (*dans le module pickle*), 511
DEFAULT_TIMEOUT (*attribut unittest.mock.ThreadingMock*), 1800
default_timer() (*dans le module timeit*), 1894
DefaultContext (*dans le module decimal*), 364
DefaultCookiePolicy (*classe dans http.cookiejar*), 1513
defaultdict (*classe dans collections*), 268
DefaultDict (*classe dans typing*), 1719
DefaultEventLoopPolicy (*classe dans asyncio*), 1149
DefaultHandler() (*méthode xml.parsers.expat.xmlparser*), 1407

DefaultHandlerExpand() (*méthode xml.parsers.expat.xmlparser*), 1407
defaults() (*méthode configparser.ConfigParser*), 635
DefaultSelector (*classe dans selectors*), 1233
defaultTestLoader (*dans le module unittest*), 1779
defaultTestResult() (*méthode unittest.TestCase*), 1770
defects (*attribut email.headerregistry.BaseHeader*), 1273
defects (*attribut email.message.EmailMessage*), 1258
defects (*attribut email.message.Message*), 1296
defpath (*dans le module os*), 728
DefragResult (*classe dans urllib.parse*), 1449
DefragResultBytes (*classe dans urllib.parse*), 1449
degrees() (*dans le module math*), 345
degrees() (*dans le module turtle*), 1584
del
    statement, 49, 90
Del (*classe dans ast*), 2109
DEL (*dans le module curses.ascii*), 951
del_param() (*méthode email.message.EmailMessage*), 1254
del_param() (*méthode email.message.Message*), 1294
delattr()
    built-in function, 13
delay() (*dans le module turtle*), 1597
delay_output() (*dans le module curses*), 925
delayload (*attribut http.cookiejar.FileCookieJar*), 1515
delch() (*méthode curses.window*), 932
dele() (*méthode poplib.POP3*), 1473
Delete (*classe dans ast*), 2117
delete() (*méthode ftplib.FTP*), 1468
delete() (*méthode imaplib.IMAP4*), 1476
delete() (*méthode tkinter.ttk.Treeview*), 1652
DELETE_ATTR (*opcode*), 2171
DELETE_DEREF (*opcode*), 2175
DELETE_FAST (*opcode*), 2175
DELETE_GLOBAL (*opcode*), 2171
DELETE_NAME (*opcode*), 2170
DELETE_SUBSCR (*opcode*), 2167
deleteacl() (*méthode imaplib.IMAP4*), 1476
deletefilehandler() (*méthode _tkinter.Widget.tk*), 1632
DeleteKey() (*dans le module winreg*), 2187
DeleteKeyEx() (*dans le module winreg*), 2187
deleteln() (*méthode curses.window*), 932
deleteMe() (*méthode bdb.Breakpoint*), 1869
DeleteValue() (*dans le module winreg*), 2187
delimiter (*attribut csv.Dialect*), 618
delitem() (*dans le module operator*), 440
deliver_challenge() (*dans le module multiprocessing.connection*), 1001
delocalize() (*dans le module locale*), 1567
demo_app() (*dans le module wsgiref.simple_server*), 1419
denominator (*attribut fractions.Fraction*), 381
denominator (*attribut numbers.Rational*), 336
deprecated() (*dans le module warnings*), 1977
DeprecationWarning, 116
deque (*classe dans collections*), 264
Deque (*classe dans typing*), 1720
dequeue() (*méthode logging.handlers.QueueListener*), 799
DER_cert_to_PEM_cert() (*dans le module ssl*), 1194
derive() (*méthode BaseExceptionGroup*), 118
derwin() (*méthode curses.window*), 933
descripteur, **2233**
description (*attribut inspect.Parameter.kind*), 2035
description (*attribut sqlite3.Cursor*), 554
deserialize() (*méthode sqlite3.Connection*), 551
dest (*attribut optparse.Option*), 907
detach() (*méthode io.BufferedIOBase*), 736
detach() (*méthode io.TextIOBase*), 740
detach() (*méthode socket.socket*), 1179
detach() (*méthode tkinter.ttk.Treeview*), 1652
detach() (*méthode weakref.finalize*), 295
Detach() (*méthode winreg.PyHKEY*), 2194
DETACHED_PROCESS (*dans le module subprocess*), 1042
--details
    option de ligne de commande inspect, 2044
details (*attribut ctypes.COMError*), 848
detect_api_mismatch() (*dans le module test.support*), 1854
detect_encoding() (*dans le module tokenize*), 2148
deterministic profiling, 1885
dev_mode (*attribut sys.flags*), 1936
device_encoding() (*dans le module os*), 674
devmajor (*attribut tarfile.TarInfo*), 604
devminor (*attribut tarfile.TarInfo*), 604
devnull (*dans le module os*), 728
DEVNULL (*dans le module subprocess*), 1030
devpoll() (*dans le module select*), 1224
DevpollSelector (*classe dans selectors*), 1233
dgettext() (*dans le module gettext*), 1554
dgettext() (*dans le module locale*), 1569
dialect (*attribut csv.csvreader*), 618
dialect (*attribut csv.csvwriter*), 619
Dialect (*classe dans csv*), 616
Dialog (*classe dans tkinter.commondialog*), 1637
Dialog (*classe dans tkinter.simpledialog*), 1634
Dict (*classe dans ast*), 2108
Dict (*classe dans typing*), 1719
dict (*classe de base*), 90
dict() (*méthode multiprocessing.managers.SyncManager*), 993
DICT_MERGE (*opcode*), 2173
DICT_UPDATE (*opcode*), 2172
DictComp (*classe dans ast*), 2113
dictConfig() (*dans le module logging.config*), 774
dictionary
    object, 90
    type, operations on, 90
dictionnaire, **2233**
dictionnaire en compréhension (*ou dictionnaire en intension*), **2233**

DictReader (*classe dans csv*), 615
DictWriter (*classe dans csv*), 615
diff_bytes() (*dans le module difflib*), 160
diff_files (*attribut filecmp.dircmp*), 486
Differ (*classe dans difflib*), 156
difference() (*méthode frozenset*), 89
difference_update() (*méthode frozenset*), 90
difflib
    module, 155
dig (*attribut sys.float_info*), 1938
digest() (*dans le module hmac*), 657
digest() (*méthode hashlib.hash*), 648
digest() (*méthode hashlib.shake*), 648
digest() (*méthode hmac.HMAC*), 657
digest_size (*attribut hmac.HMAC*), 658
digit() (*dans le module unicodedata*), 172
digits (*dans le module string*), 121
dir()
    built-in function, 13
dir() (*méthode ftplib.FTP*), 1468
dircmp (*classe dans filecmp*), 485
directory
    changing, 687
    creating, 692
    deleting, 500, 694
    option de ligne de commande compileall, 2156
    site-packages, 2049
    traversal, 704, 705
    walking, 704, 705
Directory (*classe dans tkinter.filedialog*), 1635
DirEntry (*classe dans os*), 696
dirname() (*dans le module os.path*), 473
dirs_double_event() (*méthode tkinter.filedialog.FileDialog*), 1636
dirs_select_event() (*méthode tkinter.filedialog.FileDialog*), 1636
DirsOnSysPath (*classe dans test.support.import_helper*), 1862
DIRTYPE (*dans le module tarfile*), 598
dis
    module, 2160
dis() (*dans le module dis*), 2162
dis() (*dans le module pickletools*), 2182
dis() (*méthode dis.Bytecode*), 2161
DISABLE (*dans le module sys.monitoring*), 1958
disable (*pdb command*), 1880
disable() (*dans le module faulthandler*), 1875
disable() (*dans le module gc*), 2022
disable() (*dans le module logging*), 770
disable() (*méthode bdb.Breakpoint*), 1869
disable() (*méthode profile.Profile*), 1889
disable_faulthandler() (*dans le module test.support*), 1851
disable_gc() (*dans le module test.support*), 1851
disable_interspersed_args() (*méthode optparse.OptionParser*), 912
disabled (*attribut logging.Logger*), 758
DisableReflectionKey() (*dans le module winreg*), 2191
disassemble() (*dans le module dis*), 2163
discard (*attribut http.cookiejar.Cookie*), 1519
discard() (*méthode frozenset*), 90
discard() (*méthode mailbox.Mailbox*), 1318
discard() (*méthode mailbox.MH*), 1324
disco() (*dans le module dis*), 2163
discover() (*méthode unittest.TestLoader*), 1775
disk_usage() (*dans le module shutil*), 501
dispatch_call() (*méthode bdb.Bdb*), 1871
dispatch_exception() (*méthode bdb.Bdb*), 1871
dispatch_line() (*méthode bdb.Bdb*), 1871
dispatch_return() (*méthode bdb.Bdb*), 1871
dispatch_table (*attribut pickle.Pickler*), 513
DISPLAY, 1621
display (*pdb command*), 1882
display_name (*attribut email.headerregistry.Address*), 1278
display_name (*attribut email.headerregistry.Group*), 1278
displayhook() (*dans le module sys*), 1932
dist() (*dans le module math*), 344
distance() (*dans le module turtle*), 1584
distb() (*dans le module dis*), 2163
Distribution (*classe dans importlib.metadata*), 2094
distribution simple, **2243**
distribution() (*dans le module importlib.metadata*), 2094
distutils
    module, 2222
Div (*classe dans ast*), 2110
divide() (*méthode decimal.Context*), 366
divide_int() (*méthode decimal.Context*), 366
division entière, **2234**
DivisionByZero (*classe dans decimal*), 370
divmod()
    built-in function, 14
divmod() (*méthode decimal.Context*), 367
DLE (*dans le module curses.ascii*), 951
DllCanUnloadNow() (*dans le module ctypes*), 840
DllGetClassObject() (*dans le module ctypes*), 840
dllhandle (*dans le module sys*), 1932
dllist() (*dans le module ctypes.util*), 840
dnd_start() (*dans le module tkinter.dnd*), 1640
DndHandler (*classe dans tkinter.dnd*), 1640
dngettext() (*dans le module gettext*), 1554
dnpgettext() (*dans le module gettext*), 1554
do_clear() (*méthode bdb.Bdb*), 1872
do_command() (*méthode curses.textpad.Textbox*), 949
do_GET() (*méthode http.server.SimpleHTTPRequestHandler*), 1506
do_handshake() (*méthode ssl.SSLSocket*), 1203
do_HEAD() (*méthode http.server.SimpleHTTPRequestHandler*), 1506
do_help() (*méthode cmd.Cmd*), 1609

do_POST() (*méthode http.server.CGIHTTPRequestHandler*), 1508
doc (*attribut json.JSONDecodeError*), 1314
doc (*attribut tomllib.TOMLDecodeError*), 640
doc_header (*attribut cmd.Cmd*), 1610
DocCGIXMLRPCRequestHandler (*classe dans xmlrpc.server*), 1533
DocFileSuite() (*dans le module doctest*), 1741
doClassCleanups() (*méthode de la classe unittest.TestCase*), 1771
doCleanups() (*méthode unittest.TestCase*), 1771
docmd() (*méthode smtplib.SMTP*), 1484
docstring (*attribut doctest.DocTest*), 1744
doctest
    module, 1728
DocTest (*classe dans doctest*), 1743
DocTestFailure, 1750
DocTestFinder (*classe dans doctest*), 1744
DocTestParser (*classe dans doctest*), 1745
DocTestRunner (*classe dans doctest*), 1746
DocTestSuite() (*dans le module doctest*), 1742
doctype() (*méthode xml.etree.ElementTree.TreeBuilder*), 1371
documentation
    generation, 1724
    online, 1724
documentElement (*attribut xml.dom.Document*), 1379
DocXMLRPCRequestHandler (*classe dans xmlrpc.server*), 1533
DocXMLRPCServer (*classe dans xmlrpc.server*), 1533
domain (*attribut email.headerregistry.Address*), 1278
domain (*attribut http.cookiejar.Cookie*), 1519
domain (*attribut http.cookies.Morsel*), 1510
domain (*attribut tracemalloc.DomainFilter*), 1907
domain (*attribut tracemalloc.Filter*), 1907
domain (*attribut tracemalloc.Trace*), 1910
domain_initial_dot (*attribut http.cookiejar.Cookie*), 1520
domain_return_ok() (*méthode http.cookiejar.CookiePolicy*), 1516
domain_specified (*attribut http.cookiejar.Cookie*), 1520
DomainFilter (*classe dans tracemalloc*), 1907
DomainLiberal (*attribut http.cookiejar.DefaultCookiePolicy*), 1519
DomainRFC2965Match (*attribut http.cookiejar.DefaultCookiePolicy*), 1519
DomainStrict (*attribut http.cookiejar.DefaultCookiePolicy*), 1519
DomainStrictNoDots (*attribut http.cookiejar.DefaultCookiePolicy*), 1518
DomainStrictNonDomain (*attribut http.cookiejar.DefaultCookiePolicy*), 1518
DOMEventStream (*classe dans xml.dom.pulldom*), 1390
DOMException, 1382
doModuleCleanups() (*dans le module unittest*), 1782
DomstringSizeErr, 1382
done() (*dans le module turtle*), 1599
done() (*méthode asyncio.Future*), 1132
done() (*méthode asyncio.Task*), 1081
done() (*méthode concurrent.futures.Future*), 1026
done() (*méthode graphlib.TopologicalSorter*), 332
DONT_ACCEPT_BLANKLINE (*dans le module doctest*), 1735
DONT_ACCEPT_TRUE_FOR_1 (*dans le module doctest*), 1735
dont_write_bytecode (*attribut sys.flags*), 1936
dont_write_bytecode (*dans le module sys*), 1933
doRollover() (*méthode logging.handlers.RotatingFileHandler*), 790
doRollover() (*méthode logging.handlers.TimedRotatingFileHandler*), 791
DOT (*dans le module token*), 2144
dot() (*dans le module turtle*), 1582
DOTALL (*dans le module re*), 141
doublequote (*attribut csv.Dialect*), 618
DOUBLESLASH (*dans le module token*), 2146
DOUBLESLASHEQUAL (*dans le module token*), 2146
DOUBLESTAR (*dans le module token*), 2145
DOUBLESTAREQUAL (*dans le module token*), 2146
doupdate() (*dans le module curses*), 925
down (*pdb command*), 1880
down() (*dans le module turtle*), 1585
dpgettext() (*dans le module gettext*), 1554
drain() (*méthode asyncio.StreamWriter*), 1088
drive (*attribut pathlib.PurePath*), 450
drop_whitespace (*attribut textwrap.TextWrapper*), 171
dropwhile() (*dans le module itertools*), 414
dst() (*méthode datetime.datetime*), 224
dst() (*méthode datetime.time*), 232
dst() (*méthode datetime.timezone*), 240
dst() (*méthode datetime.tzinfo*), 233
DTDHandler (*classe dans xml.sax.handler*), 1393
dump() (*dans le module ast*), 2137
dump() (*dans le module json*), 1310
dump() (*dans le module marshal*), 529
dump() (*dans le module pickle*), 511
dump() (*dans le module plistlib*), 643
dump() (*dans le module xml.etree.ElementTree*), 1362
dump() (*méthode pickle.Pickler*), 512
dump() (*méthode tracemalloc.Snapshot*), 1908
dump_stats() (*méthode profile.Profile*), 1889
dump_stats() (*méthode pstats.Stats*), 1890
dump_traceback() (*dans le module faulthandler*), 1874
dump_traceback_later() (*dans le module faulthandler*), 1875
dumps() (*dans le module json*), 1310
dumps() (*dans le module marshal*), 530
dumps() (*dans le module pickle*), 511
dumps() (*dans le module plistlib*), 643
dumps() (*dans le module xmlrpc.client*), 1527
dup() (*dans le module os*), 674
dup() (*méthode socket.socket*), 1179

dup2() (*dans le module os*), 674
DuplicateOptionError, 638
DuplicateSectionError, 638
--durations
    option de ligne de commande unittest, 1754
dwFlags (*attribut subprocess.STARTUPINFO*), 1039
DynamicClassAttribute() (*dans le module types*), 305

E

-e
    option de ligne de commande calendar, 257
    option de ligne de commande compileall, 2157
    option de ligne de commande tarfile, 609
    option de ligne de commande tokenize, 2149
    option de ligne de commande zipfile, 595
e (*dans le module cmath*), 350
e (*dans le module math*), 346
E2BIG (*dans le module errno*), 804
EACCES (*dans le module errno*), 805
EADDRINUSE (*dans le module errno*), 809
EADDRNOTAVAIL (*dans le module errno*), 809
EADV (*dans le module errno*), 807
EAFNOSUPPORT (*dans le module errno*), 809
EAFP, **2233**
EAGAIN (*dans le module errno*), 805
eager_task_factory() (*dans le module asyncio*), 1073
EALREADY (*dans le module errno*), 810
east_asian_width() (*dans le module unicodedata*), 173
EAUTH (*dans le module errno*), 811
EBADARCH (*dans le module errno*), 811
EBADE (*dans le module errno*), 807
EBADEXEC (*dans le module errno*), 811
EBADF (*dans le module errno*), 805
EBADFD (*dans le module errno*), 808
EBADMACHO (*dans le module errno*), 811
EBADMSG (*dans le module errno*), 808
EBADR (*dans le module errno*), 807
EBADRPC (*dans le module errno*), 812
EBADRQC (*dans le module errno*), 807
EBADSLT (*dans le module errno*), 807
EBFONT (*dans le module errno*), 807
EBUSY (*dans le module errno*), 805
ECANCELED (*dans le module errno*), 812
ECHILD (*dans le module errno*), 805
echo() (*dans le module curses*), 925
echochar() (*méthode curses.window*), 933
ECHRNG (*dans le module errno*), 806
ECOMM (*dans le module errno*), 807
ECONNABORTED (*dans le module errno*), 809
ECONNREFUSED (*dans le module errno*), 809
ECONNRESET (*dans le module errno*), 809
EDEADLK (*dans le module errno*), 806
EDEADLOCK (*dans le module errno*), 807
EDESTADDRREQ (*dans le module errno*), 808
EDEVERR (*dans le module errno*), 811
edit() (*méthode curses.textpad.Textbox*), 948
EDOM (*dans le module errno*), 806
EDOTDOT (*dans le module errno*), 808
EDQUOT (*dans le module errno*), 810
EEXIST (*dans le module errno*), 805
EFAULT (*dans le module errno*), 805
EFBIG (*dans le module errno*), 805
EFD_CLOEXEC (*dans le module os*), 708
EFD_NONBLOCK (*dans le module os*), 708
EFD_SEMAPHORE (*dans le module os*), 708
effective() (*dans le module bdb*), 1873
EFTYPE (*dans le module errno*), 811
ehlo() (*méthode smtplib.SMTP*), 1484
ehlo_or_helo_if_needed() (*méthode smtplib.SMTP*), 1484
EHOSTDOWN (*dans le module errno*), 810
EHOSTUNREACH (*dans le module errno*), 810
EHWPOISON (*dans le module errno*), 810
EIDRM (*dans le module errno*), 806
EILSEQ (*dans le module errno*), 808
EINPROGRESS (*dans le module errno*), 810
EINTR (*dans le module errno*), 804
EINVAL (*dans le module errno*), 805
EIO (*dans le module errno*), 804
EISCONN (*dans le module errno*), 809
EISDIR (*dans le module errno*), 805
EISNAM (*dans le module errno*), 810
EJECT (*attribut enum.FlagBoundary*), 327
EKEYEXPIRED (*dans le module errno*), 810
EKEYREJECTED (*dans le module errno*), 810
EKEYREVOKED (*dans le module errno*), 810
EL2HLT (*dans le module errno*), 807
EL2NSYNC (*dans le module errno*), 806
EL3HLT (*dans le module errno*), 806
EL3RST (*dans le module errno*), 806
Element (*classe dans xml.etree.ElementTree*), 1366
element_create() (*méthode tkinter.ttk.Style*), 1656
element_names() (*méthode tkinter.ttk.Style*), 1658
element_options() (*méthode tkinter.ttk.Style*), 1658
ElementDeclHandler() (*méthode xml.parsers.expat.xmlparser*), 1406
elements() (*méthode collections.Counter*), 262
ElementTree (*classe dans xml.etree.ElementTree*), 1369
ELIBACC (*dans le module errno*), 808
ELIBBAD (*dans le module errno*), 808
ELIBEXEC (*dans le module errno*), 808
ELIBMAX (*dans le module errno*), 808
ELIBSCN (*dans le module errno*), 808
ELLIPSIS (*dans le module doctest*), 1736
ELLIPSIS (*dans le module token*), 2146
Ellipsis (*variable de base*), 36
EllipsisType (*dans le module types*), 303

ELNRNG (*dans le module errno*), 806
ELOCKUNMAPPED (*dans le module errno*), 811
ELOOP (*dans le module errno*), 806
EM (*dans le module curses.ascii*), 951
email
    module, 1249
email.charset
    module, 1301
email.contentmanager
    module, 1279
email.encoders
    module, 1303
email.errors
    module, 1272
email.generator
    module, 1262
email.header
    module, 1298
email.headerregistry
    module, 1273
email.iterators
    module, 1306
email.message
    module, 1250
EmailMessage (*classe dans email.message*), 1251
email.mime
    module, 1296
email.mime.application
    module, 1297
email.mime.audio
    module, 1297
email.mime.base
    module, 1296
email.mime.image
    module, 1297
email.mime.message
    module, 1298
email.mime.multipart
    module, 1296
email.mime.nonmultipart
    module, 1296
email.mime.text
    module, 1298
email.parser
    module, 1258
email.policy
    module, 1264
EmailPolicy (*classe dans email.policy*), 1269
email.utils
    module, 1304
EMEDIUMTYPE (*dans le module errno*), 810
EMFILE (*dans le module errno*), 805
emit() (*méthode logging.FileHandler*), 787
emit() (*méthode logging.Handler*), 763
emit() (*méthode logging.handlers.BufferingHandler*), 796
emit() (*méthode logging.handlers.DatagramHandler*), 792
emit() (*méthode logging.handlers.HTTPHandler*), 797
emit() (*méthode logging.handlers.NTEventLogHandler*), 795
emit() (*méthode logging.handlers.QueueHandler*), 798
emit() (*méthode logging.handlers.RotatingFileHandler*), 790
emit() (*méthode logging.handlers.SMTPHandler*), 796
emit() (*méthode logging.handlers.SocketHandler*), 791
emit() (*méthode logging.handlers.SysLogHandler*), 793
emit() (*méthode logging.handlers.TimedRotatingFileHandler*), 791
emit() (*méthode logging.handlers.WatchedFileHandler*), 788
emit() (*méthode logging.NullHandler*), 787
emit() (*méthode logging.StreamHandler*), 786
EMLINK (*dans le module errno*), 806
Empty, 1049
empty (*attribut inspect.Parameter*), 2034
empty (*attribut inspect.Signature*), 2032
empty() (*méthode asyncio.Queue*), 1102
empty() (*méthode multiprocessing.Queue*), 980
empty() (*méthode multiprocessing.SimpleQueue*), 982
empty() (*méthode queue.Queue*), 1050
empty() (*méthode queue.SimpleQueue*), 1052
empty() (*méthode sched.scheduler*), 1048
EMPTY_NAMESPACE (*dans le module xml.dom*), 1375
emptyline() (*méthode cmd.Cmd*), 1609
emscripten_version (*attribut sys._emscripten_info*), 1933
EMSGSIZE (*dans le module errno*), 808
EMULTIHOP (*dans le module errno*), 808
enable (*pdb command*), 1880
enable() (*dans le module faulthandler*), 1875
enable() (*dans le module gc*), 2021
enable() (*méthode bdb.Breakpoint*), 1869
enable() (*méthode imaplib.IMAP4*), 1476
enable() (*méthode profile.Profile*), 1889
enable_callback_tracebacks() (*dans le module sqlite3*), 540
enable_interspersed_args() (*méthode optparse.OptionParser*), 912
enable_load_extension() (*méthode sqlite3.Connection*), 547
enable_traversal() (*méthode tkinter.ttk.Notebook*), 1647
ENABLE_USER_SITE (*dans le module site*), 2051
enabled (*attribut bdb.Breakpoint*), 1870
EnableReflectionKey() (*dans le module winreg*), 2191
ENAMETOOLONG (*dans le module errno*), 806
ENAVAIL (*dans le module errno*), 810
enclose() (*méthode curses.window*), 933
encodage du système de fichiers et gestionnaire d'erreurs associé, **2234**
encodage régional, **2238**
encodages de texte, **2243**

encode
    Codecs, 189
encode (*attribut codecs.CodecInfo*), 189
encode() (*dans le module base64*), 1342
encode() (*dans le module codecs*), 189
encode() (*dans le module quopri*), 1345
encode() (*méthode codecs.Codec*), 194
encode() (*méthode codecs.IncrementalEncoder*), 195
encode() (*méthode email.header.Header*), 1300
encode() (*méthode json.JSONEncoder*), 1314
encode() (*méthode str*), 55
encode() (*méthode xmlrpc.client.Binary*), 1524
encode() (*méthode xmlrpc.client.DateTime*), 1524
encode_7or8bit() (*dans le module email.encoders*), 1304
encode_base64() (*dans le module email.encoders*), 1303
encode_noop() (*dans le module email.encoders*), 1304
encode_quopri() (*dans le module email.encoders*), 1303
encode_rfc2231() (*dans le module email.utils*), 1306
encodebytes() (*dans le module base64*), 1342
EncodedFile() (*dans le module codecs*), 191
encodePriority() (*méthode logging.handlers.SysLogHandler*), 794
encodestring() (*dans le module quopri*), 1345
encoding
    base64, 1339
    quoted-printable, 1345
--encoding
    option de ligne de commande calendar, 257
encoding (*attribut curses.window*), 933
encoding (*attribut io.TextIOBase*), 740
encoding (*attribut UnicodeError*), 114
ENCODING (*dans le module tarfile*), 598
ENCODING (*dans le module token*), 2146
encodings_map (*attribut mimetypes.MimeTypes*), 1338
encodings_map (*dans le module mimetypes*), 1337
encodings.idna
    module, 204
encodings.mbcs
    module, 205
encodings.utf_8_sig
    module, 205
EncodingWarning, 117
end (*attribut UnicodeError*), 114
end() (*méthode re.Match*), 149
end() (*méthode xml.etree.ElementTree.TreeBuilder*), 1371
END_ASYNC_FOR (*opcode*), 2168
end_col_offset (*attribut ast.AST*), 2105
end_fill() (*dans le module turtle*), 1588
END_FOR (*opcode*), 2166
end_headers() (*méthode http.server.BaseHTTPRequestHandler*), 1505
end_lineno (*attribut ast.AST*), 2105
end_lineno (*attribut SyntaxError*), 113
end_lineno (*attribut traceback.TracebackException*), 2014
end_ns() (*méthode xml.etree.ElementTree.TreeBuilder*), 1371
end_offset (*attribut SyntaxError*), 113
end_offset (*attribut traceback.TracebackException*), 2014
end_poly() (*dans le module turtle*), 1593
END_SEND (*opcode*), 2166
endCDATA() (*méthode xml.sax.handler.LexicalHandler*), 1398
EndCdataSectionHandler() (*méthode xml.parsers.expat.xmlparser*), 1407
EndDoctypeDeclHandler() (*méthode xml.parsers.expat.xmlparser*), 1406
endDocument() (*méthode xml.sax.handler.ContentHandler*), 1395
endDTD() (*méthode xml.sax.handler.LexicalHandler*), 1397
endElement() (*méthode xml.sax.handler.ContentHandler*), 1395
EndElementHandler() (*méthode xml.parsers.expat.xmlparser*), 1407
endElementNS() (*méthode xml.sax.handler.ContentHandler*), 1396
endheaders() (*méthode http.client.HTTPConnection*), 1462
ENDMARKER (*dans le module token*), 2143
EndNamespaceDeclHandler() (*méthode xml.parsers.expat.xmlparser*), 1407
endpos (*attribut re.Match*), 150
endPrefixMapping() (*méthode xml.sax.handler.ContentHandler*), 1395
endswith() (*méthode bytearray*), 69
endswith() (*méthode bytes*), 69
endswith() (*méthode str*), 55
endwin() (*dans le module curses*), 925
ENEEDAUTH (*dans le module errno*), 811
ENETDOWN (*dans le module errno*), 809
ENETRESET (*dans le module errno*), 809
ENETUNREACH (*dans le module errno*), 809
ENFILE (*dans le module errno*), 805
ENOANO (*dans le module errno*), 807
ENOATTR (*dans le module errno*), 811
ENOBUFS (*dans le module errno*), 809
ENOCSI (*dans le module errno*), 806
ENODATA (*dans le module errno*), 807
ENODEV (*dans le module errno*), 805
ENOENT (*dans le module errno*), 804
ENOEXEC (*dans le module errno*), 804
ENOKEY (*dans le module errno*), 810
ENOLCK (*dans le module errno*), 806
ENOLINK (*dans le module errno*), 807
ENOMEDIUM (*dans le module errno*), 810
ENOMEM (*dans le module errno*), 805
ENOMSG (*dans le module errno*), 806
ENONET (*dans le module errno*), 807

ENOPKG (*dans le module errno*), 807
ENOPOLICY (*dans le module errno*), 811
ENOPROTOOPT (*dans le module errno*), 808
ENOSPC (*dans le module errno*), 805
ENOSR (*dans le module errno*), 807
ENOSTR (*dans le module errno*), 807
ENOSYS (*dans le module errno*), 806
ENOTACTIVE (*dans le module errno*), 811
ENOTBLK (*dans le module errno*), 805
ENOTCAPABLE (*dans le module errno*), 812
ENOTCONN (*dans le module errno*), 809
ENOTDIR (*dans le module errno*), 805
ENOTEMPTY (*dans le module errno*), 806
ENOTNAM (*dans le module errno*), 810
ENOTRECOVERABLE (*dans le module errno*), 812
ENOTSOCK (*dans le module errno*), 808
ENOTSUP (*dans le module errno*), 809
ENOTTY (*dans le module errno*), 805
ENOTUNIQ (*dans le module errno*), 808
ENQ (*dans le module curses.ascii*), 950
enqueue() (*méthode logging.handlers.QueueHandler*), 798
enqueue_sentinel() (*méthode logging.handlers.QueueListener*), 799
ensemble en compréhension (*ou ensemble en intension*), **2243**
ensure_directories() (*méthode venv.EnvBuilder*), 1919
ensure_future() (*dans le module asyncio*), 1131
ensurepip
    module, 1913
enter() (*méthode sched.scheduler*), 1048
enter_async_context() (*méthode contextlib.AsyncExitStack*), 1998
enter_context() (*méthode contextlib.ExitStack*), 1997
enterabs() (*méthode sched.scheduler*), 1048
enterAsyncContext() (*méthode unittest.IsolatedAsyncioTestCase*), 1772
enterClassContext() (*méthode de la classe unittest.TestCase*), 1771
enterContext() (*méthode unittest.TestCase*), 1771
enterModuleContext() (*dans le module unittest*), 1782
entities (*attribut xml.dom.DocumentType*), 1379
EntityDeclHandler() (*méthode xml.parsers.expat.xmlparser*), 1407
entitydefs (*dans le module html.entities*), 1352
EntityResolver (*classe dans xml.sax.handler*), 1393
entrée de chemin, **2241**
entry_points() (*dans le module importlib.metadata*), 2091
EntryPoint (*classe dans importlib.metadata*), 2091
EntryPoints (*classe dans importlib.metadata*), 2091
enum
    module, 316
Enum (*classe dans enum*), 319
enum_certificates() (*dans le module ssl*), 1194
enum_crls() (*dans le module ssl*), 1195
EnumCheck (*classe dans enum*), 326
EnumDict (*classe dans enum*), 328
enumerate()
    built-in function, 14
enumerate() (*dans le module threading*), 957
EnumKey() (*dans le module winreg*), 2187
EnumType (*classe dans enum*), 318
EnumValue() (*dans le module winreg*), 2188
EnvBuilder (*classe dans venv*), 1918
environ (*dans le module os*), 665
environ (*dans le module posix*), 2198
environb (*dans le module os*), 666
environment variables
    deleting, 672
    setting, 669
EnvironmentError, 114
Environments
    virtual, 1915
EnvironmentVarGuard (*classe dans test.support.os_helper*), 1859
environnement virtuel, **2244**
ENXIO (*dans le module errno*), 804
eof (*attribut bz2.BZ2Decompressor*), 577
eof (*attribut lzma.LZMADecompressor*), 582
eof (*attribut shlex.shlex*), 1616
eof (*attribut ssl.MemoryBIO*), 1221
eof (*attribut zlib.Decompress*), 570
eof_received() (*méthode asyncio.BufferedProtocol*), 1141
eof_received() (*méthode asyncio.Protocol*), 1140
EOFError, 109
EOPNOTSUPP (*dans le module errno*), 809
EOT (*dans le module curses.ascii*), 950
EOVERFLOW (*dans le module errno*), 808
EOWNERDEAD (*dans le module errno*), 812
EPERM (*dans le module errno*), 804
EPFNOSUPPORT (*dans le module errno*), 809
epilogue (*attribut email.message.EmailMessage*), 1258
epilogue (*attribut email.message.Message*), 1296
EPIPE (*dans le module errno*), 806
epoch, 744
epoll() (*dans le module select*), 1224
EpollSelector (*classe dans selectors*), 1233
EPROCLIM (*dans le module errno*), 811
EPROCUNAVAIL (*dans le module errno*), 811
EPROGMISMATCH (*dans le module errno*), 811
EPROGUNAVAIL (*dans le module errno*), 811
EPROTO (*dans le module errno*), 807
EPROTONOSUPPORT (*dans le module errno*), 809
EPROTOTYPE (*dans le module errno*), 808
epsilon (*attribut sys.float_info*), 1938
EPWROFF (*dans le module errno*), 812
Eq (*classe dans ast*), 2111
eq() (*dans le module operator*), 437
EQEQUAL (*dans le module token*), 2145
EQFULL (*dans le module errno*), 810
EQUAL (*dans le module token*), 2144

ERA (*dans le module locale*), 1564
ERA_D_FMT (*dans le module locale*), 1565
ERA_D_T_FMT (*dans le module locale*), 1565
ERA_T_FMT (*dans le module locale*), 1565
ERANGE (*dans le module errno*), 806
erase() (*méthode curses.window*), 933
erasechar() (*dans le module curses*), 925
EREMCHG (*dans le module errno*), 808
EREMOTE (*dans le module errno*), 807
EREMOTEIO (*dans le module errno*), 810
ERESTART (*dans le module errno*), 808
erf() (*dans le module math*), 346
erfc() (*dans le module math*), 346
ERFKILL (*dans le module errno*), 810
EROFS (*dans le module errno*), 806
ERPCMISMATCH (*dans le module errno*), 812
ERR (*dans le module curses*), 937
errcheck (*attribut ctypes._CFuncPtr*), 836
errcode (*attribut xmlrpc.client.ProtocolError*), 1526
errmsg (*attribut xmlrpc.client.ProtocolError*), 1526
errno
    module, 111, 804
errno (*attribut OSError*), 111
Error, 306, 502, 556, 617, 638, 1334, 1344, 1414, 1549, 1561
error, 182, 531, 533535, 567, 663, 924, 1057, 1165, 1224, 1403, 2206, 2218
ERROR (*dans le module logging*), 762
ERROR (*dans le module tkinter.messagebox*), 1639
error handler's name
    backslashreplace, 192
    ignore, 192
    namereplace, 192
    replace, 192
    strict, 192
    surrogateescape, 192
    surrogatepass, 192
    xmlcharrefreplace, 192
error() (*dans le module logging*), 770
error() (*méthode argparse.ArgumentParser*), 879
error() (*méthode logging.Logger*), 761
error() (*méthode urllib.request.OpenerDirector*), 1433
error() (*méthode xml.sax.handler.ErrorHandler*), 1397
error_body (*attribut wsgiref.handlers.BaseHandler*), 1423
error_content_type (*attribut http.server.BaseHTTPRequestHandler*), 1504
error_headers (*attribut wsgiref.handlers.BaseHandler*), 1423
error_leader() (*méthode shlex.shlex*), 1615
error_message_format (*attribut http.server.BaseHTTPRequestHandler*), 1504
error_output() (*méthode wsgiref.handlers.BaseHandler*), 1423
error_perm, 1470
error_proto, 1470, 1472
error_received() (*méthode asyncio.DatagramProtocol*), 1141
error_reply, 1470
error_status (*attribut wsgiref.handlers.BaseHandler*), 1423
error_temp, 1470
ErrorByteIndex (*attribut xml.parsers.expat.xmlparser*), 1406
ErrorCode (*attribut xml.parsers.expat.xmlparser*), 1406
errorcode (*dans le module errno*), 804
ErrorColumnNumber (*attribut xml.parsers.expat.xmlparser*), 1406
ErrorHandler (*classe dans xml.sax.handler*), 1393
errorlevel (*attribut tarfile.TarFile*), 602
ErrorLineNumber (*attribut xml.parsers.expat.xmlparser*), 1406
Errors
    logging, 756
errors (*attribut io.TextIOBase*), 740
errors (*attribut unittest.TestLoader*), 1774
errors (*attribut unittest.TestResult*), 1776
ErrorStream (*classe dans wsgiref.types*), 1424
ErrorString() (*dans le module xml.parsers.expat*), 1403
ERRORTOKEN (*dans le module token*), 2146
ESC (*dans le module curses.ascii*), 951
escape (*attribut shlex.shlex*), 1616
escape() (*dans le module glob*), 493
escape() (*dans le module html*), 1347
escape() (*dans le module re*), 145
escape() (*dans le module xml.sax.saxutils*), 1398
escapechar (*attribut csv.Dialect*), 618
escapedquotes (*attribut shlex.shlex*), 1616
ESHLIBVERS (*dans le module errno*), 812
ESHUTDOWN (*dans le module errno*), 809
ESOCKTNOSUPPORT (*dans le module errno*), 809
espace
    in string formatting, 125
espace de nommage, **2239**
ESPIPE (*dans le module errno*), 805
ESRCH (*dans le module errno*), 804
ESRMNT (*dans le module errno*), 807
ESTALE (*dans le module errno*), 810
ESTRPIPE (*dans le module errno*), 808
ETB (*dans le module curses.ascii*), 951
ETH_P_ALL (*dans le module socket*), 1168
ETHERTYPE_ARP (*dans le module socket*), 1170
ETHERTYPE_IP (*dans le module socket*), 1170
ETHERTYPE_IPV6 (*dans le module socket*), 1170
ETHERTYPE_VLAN (*dans le module socket*), 1170
ETIME (*dans le module errno*), 807
ETIMEDOUT (*dans le module errno*), 809
Etiny() (*méthode decimal.Context*), 366
ETOOMANYREFS (*dans le module errno*), 809
Etop() (*méthode decimal.Context*), 366
ETX (*dans le module curses.ascii*), 950
ETXTBSY (*dans le module errno*), 805

EUCLEAN (*dans le module errno*), 810
EUNATCH (*dans le module errno*), 806
EUSERS (*dans le module errno*), 808
eval
    built-in function, 103, 308, 309
eval()
    built-in function, 14
evaluate function, **2234**
evaluate() (*méthode annotationlib.ForwardRef*), 2046
evaluate_bound() (*méthode typing.TypeVar*), 1697
evaluate_constraints() (*méthode typing.TypeVar*), 1697
evaluate_default() (*méthode typing.ParamSpec*), 1701
evaluate_default() (*méthode typing.TypeVar*), 1697
evaluate_default() (*méthode typing.TypeVarTuple*), 1699
evaluate_forward_ref() (*dans le module typing*), 1717
evaluate_value() (*méthode typing.TypeAliasType*), 1702
Event (*classe dans asyncio*), 1093
Event (*classe dans multiprocessing*), 986
Event (*classe dans threading*), 966
event scheduling, 1047
Event() (*méthode multiprocessing.managers.SyncManager*), 993
EVENT_READ (*dans le module selectors*), 1232
EVENT_WRITE (*dans le module selectors*), 1232
eventfd() (*dans le module os*), 707
eventfd_read() (*dans le module os*), 707
eventfd_write() (*dans le module os*), 708
EventLoop (*classe dans asyncio*), 1128
events (*attribut selectors.SelectorKey*), 1232
events (*widgets*), 1630
EWOULDBLOCK (*dans le module errno*), 806
EX_CANTCREAT (*dans le module os*), 714
EX_CONFIG (*dans le module os*), 714
EX_DATAERR (*dans le module os*), 713
EX_IOERR (*dans le module os*), 714
EX_NOHOST (*dans le module os*), 714
EX_NOINPUT (*dans le module os*), 714
EX_NOPERM (*dans le module os*), 714
EX_NOTFOUND (*dans le module os*), 715
EX_NOUSER (*dans le module os*), 714
EX_OK (*dans le module os*), 713
EX_OSERR (*dans le module os*), 714
EX_OSFILE (*dans le module os*), 714
EX_PROTOCOL (*dans le module os*), 714
EX_SOFTWARE (*dans le module os*), 714
EX_TEMPFAIL (*dans le module os*), 714
EX_UNAVAILABLE (*dans le module os*), 714
EX_USAGE (*dans le module os*), 713
--exact
    option de ligne de commande tokenize, 2149
EXACT_TOKEN_TYPES (*dans le module token*), 2147
example (*attribut doctest.DocTestFailure*), 1750
example (*attribut doctest.UnexpectedException*), 1750
Example (*classe dans doctest*), 1744
examples (*attribut doctest.DocTest*), 1743
exc_info (*attribut doctest.UnexpectedException*), 1750
exc_info() (*dans le module sys*), 1934
exc_msg (*attribut doctest.Example*), 1744
exc_type (*attribut traceback.TracebackException*), 2014
exc_type_str (*attribut traceback.TracebackException*), 2014
excel (*classe dans csv*), 616
excel_tab (*classe dans csv*), 616
except
    statement, 107
ExceptHandler (*classe dans ast*), 2122
excepthook() (*dans le module sys*), 1933
excepthook() (*dans le module threading*), 956
Exception, 109
exception
    chaining, 107
EXCEPTION (*dans le module _tkinter*), 1632
exception() (*dans le module logging*), 770
exception() (*dans le module sys*), 1934
exception() (*méthode asyncio.Future*), 1133
exception() (*méthode asyncio.Task*), 1081
exception() (*méthode concurrent.futures.Future*), 1026
exception() (*méthode logging.Logger*), 761
EXCEPTION_HANDLED (*monitoring event*), 1956
ExceptionGroup, 117
exceptions (*attribut BaseExceptionGroup*), 117
exceptions (*attribut traceback.TracebackException*), 2013
exceptions (*pdb command*), 1884
EXCLAMATION (*dans le module token*), 2146
EXDEV (*dans le module errno*), 805
exec
    built-in function, 15, 103
exec()
    built-in function, 15
exec_module() (*méthode importlib.abc.InspectLoader*), 2071
exec_module() (*méthode importlib.abc.Loader*), 2069
exec_module() (*méthode importlib.abc.SourceLoader*), 2072
exec_module() (*méthode importlib.machinery.ExtensionFileLoader*), 2077
exec_module() (*méthode zipimport.zipimporter*), 2058
exec_prefix (*dans le module sys*), 1934
execl() (*dans le module os*), 712
execle() (*dans le module os*), 712
execlp() (*dans le module os*), 712
execlpe() (*dans le module os*), 712
executable (*dans le module sys*), 1935
Executable Zip Files, 1924
execute() (*méthode sqlite3.Connection*), 543

execute() (*méthode sqlite3.Cursor*), 552
executemany() (*méthode sqlite3.Connection*), 543
executemany() (*méthode sqlite3.Cursor*), 552
executescript() (*méthode sqlite3.Connection*), 543
executescript() (*méthode sqlite3.Cursor*), 553
ExecutionFailed, 1028
ExecutionLoader (*classe dans importlib.abc*), 2071
Executor (*classe dans concurrent.futures*), 1020
execv() (*dans le module os*), 712
execve() (*dans le module os*), 712
execvp() (*dans le module os*), 712
execvpe() (*dans le module os*), 712
EXFULL (*dans le module errno*), 807
exists() (*dans le module os.path*), 473
exists() (*méthode pathlib.Path*), 460
exists() (*méthode pathlib.types.PathInfo*), 472
exists() (*méthode tkinter.ttk.Treeview*), 1652
exists() (*méthode zipfile.Path*), 591
exit (*variable de base*), 36
exit() (*dans le module _thread*), 1057
exit() (*dans le module sys*), 1935
exit() (*méthode argparse.ArgumentParser*), 878
exitcode (*attribut multiprocessing.Process*), 977
exitonclick() (*dans le module turtle*), 1601
ExitStack (*classe dans contextlib*), 1996
exp() (*dans le module cmath*), 348
exp() (*dans le module math*), 343
exp() (*méthode decimal.Context*), 367
exp() (*méthode decimal.Decimal*), 358
exp2() (*dans le module math*), 343
expand() (*méthode re.Match*), 148
expand_tabs (*attribut textwrap.TextWrapper*), 170
ExpandEnvironmentStrings() (*dans le module winreg*), 2188
expandNode() (*méthode xml.dom.pulldom.DOMEventStream*), 1390
expandtabs() (*méthode bytearray*), 74
expandtabs() (*méthode bytes*), 74
expandtabs() (*méthode str*), 55
expanduser() (*dans le module os.path*), 474
expanduser() (*méthode pathlib.Path*), 459
expandvars() (*dans le module os.path*), 474
Expat, 1403
ExpatError, 1403
expected (*attribut asyncio.IncompleteReadError*), 1105
expectedFailure() (*dans le module unittest*), 1760
expectedFailures (*attribut unittest.TestResult*), 1777
expired() (*méthode asyncio.Timeout*), 1075
expires (*attribut http.cookiejar.Cookie*), 1519
expires (*attribut http.cookies.Morsel*), 1510
exploded (*attribut ipaddress.IPv4Address*), 1536
exploded (*attribut ipaddress.IPv4Network*), 1541
exploded (*attribut ipaddress.IPv6Address*), 1538
exploded (*attribut ipaddress.IPv6Network*), 1544
expm1() (*dans le module math*), 343
expovariate() (*dans le module random*), 387
Expr (*classe dans ast*), 2109
expression, **2234**
Expression (*classe dans ast*), 2106
expression génératrice, **2235**
expunge() (*méthode imaplib.IMAP4*), 1476
extend() (*méthode array.array*), 290
extend() (*méthode collections.deque*), 265
extend() (*méthode xml.etree.ElementTree.Element*), 1367
extend() (*sequence method*), 49
extend_path() (*dans le module pkgutil*), 2059
EXTENDED_ARG (*opcode*), 2177
ExtendedContext (*dans le module decimal*), 364
ExtendedInterpolation (*classe dans configparser*), 626
extendleft() (*méthode collections.deque*), 265
EXTENSION_SUFFIXES (*dans le module importlib.machinery*), 2074
ExtensionFileLoader (*classe dans importlib.machinery*), 2077
extensions_map (*attribut http.server.SimpleHTTPRequestHandler*), 1506
External Data Representation, 510
external_attr (*attribut zipfile.ZipInfo*), 594
ExternalClashError, 1334
ExternalEntityParserCreate() (*méthode xml.parsers.expat.xmlparser*), 1404
ExternalEntityRefHandler() (*méthode xml.parsers.expat.xmlparser*), 1408
extra (*attribut zipfile.ZipInfo*), 593
--extract
    option de ligne de commande tarfile, 609
    option de ligne de commande zipfile, 595
extract() (*méthode de la classe traceback.StackSummary*), 2015
extract() (*méthode tarfile.TarFile*), 601
extract() (*méthode zipfile.ZipFile*), 588
extract_cookies() (*méthode http.cookiejar.CookieJar*), 1514
extract_stack() (*dans le module traceback*), 2012
extract_tb() (*dans le module traceback*), 2012
extract_version (*attribut zipfile.ZipInfo*), 594
extractall() (*méthode tarfile.TarFile*), 600
extractall() (*méthode zipfile.ZipFile*), 588
ExtractError, 597
extractfile() (*méthode tarfile.TarFile*), 601
extraction_filter (*attribut tarfile.TarFile*), 602
extsep (*dans le module os*), 728

F

-f
    option de ligne de commande calendar, 258
    option de ligne de commande compileall, 2156
    option de ligne de commande random, 392

option de ligne de commande trace, 1899
option de ligne de commande unittest, 1754
f-string, **2234**
F_CONTIGUOUS (*attribut inspect.BufferFlags*), 2044
f_contiguous (*attribut memoryview*), 88
F_LOCK (*dans le module os*), 676
F_OK (*dans le module os*), 687
F_TEST (*dans le module os*), 676
F_TLOCK (*dans le module os*), 676
F_ULOCK (*dans le module os*), 676
fabs() (*dans le module math*), 340
factorial() (*dans le module math*), 340
factory() (*méthode de la classe importlib.util.LazyLoader*), 2082
fail() (*méthode unittest.TestCase*), 1769
FAIL_FAST (*dans le module doctest*), 1737
failed (*attribut doctest.TestResults*), 1745
--failfast
option de ligne de commande unittest, 1754
failfast (*attribut unittest.TestResult*), 1777
failureException, 1742
failureException (*attribut unittest.TestCase*), 1770
failures (*attribut doctest.DocTestRunner*), 1747
failures (*attribut unittest.TestResult*), 1776
FakePath (*classe dans test.support.os_helper*), 1860
False, 37, 46
false, 37
False (*Built-in object*), 37
False (*variable de base*), 35
families() (*dans le module tkinter.font*), 1634
family (*attribut socket.socket*), 1185
--fast
option de ligne de commande gzip, 574
fast (*attribut pickle.Pickler*), 513
fatalError() (*méthode xml.sax.handler.ErrorHandler*), 1397
Fault (*classe dans xmlrpc.client*), 1525
faultCode (*attribut xmlrpc.client.Fault*), 1525
faulthandler
module, 1874
faultString (*attribut xmlrpc.client.Fault*), 1525
fchdir() (*dans le module os*), 689
fchmod() (*dans le module os*), 674
fchown() (*dans le module os*), 674
fcntl
module, 2204
fcntl() (*dans le module fcntl*), 2204
fd (*attribut selectors.SelectorKey*), 1232
fd() (*dans le module turtle*), 1578
fd_count() (*dans le module test.support.os_helper*), 1860
fdatasync() (*dans le module os*), 675
fdopen() (*dans le module os*), 673
feature_external_ges (*dans le module xml.sax.handler*), 1393
feature_external_pes (*dans le module xml.sax.handler*), 1394
feature_namespace_prefixes (*dans le module xml.sax.handler*), 1393
feature_namespaces (*dans le module xml.sax.handler*), 1393
feature_string_interning (*dans le module xml.sax.handler*), 1393
feature_validation (*dans le module xml.sax.handler*), 1393
FEBRUARY (*dans le module calendar*), 256
feed() (*méthode email.parser.BytesFeedParser*), 1259
feed() (*méthode html.parser.HTMLParser*), 1349
feed() (*méthode xml.etree.ElementTree.XMLParser*), 1372
feed() (*méthode xml.etree.ElementTree.XMLPullParser*), 1373
feed() (*méthode xml.sax.xmlreader.IncrementalParser*), 1401
feed_eof() (*méthode asyncio.StreamReader*), 1087
FeedParser (*classe dans email.parser*), 1259
fetch() (*méthode imaplib.IMAP4*), 1477
fetchall() (*méthode sqlite3.Cursor*), 553
fetchmany() (*méthode sqlite3.Cursor*), 553
fetchone() (*méthode sqlite3.Cursor*), 553
FF (*dans le module curses.ascii*), 950
fflags (*attribut select.kevent*), 1230
fichier binaire, **2231**
fichier texte, **2243**
Field (*classe dans dataclasses*), 1982
field() (*dans le module dataclasses*), 1981
field_size_limit() (*dans le module csv*), 615
fieldnames (*attribut csv.DictReader*), 619
fields (*attribut uuid.UUID*), 1489
fields() (*dans le module dataclasses*), 1982
FIFOTYPE (*dans le module tarfile*), 598
file
byte-code, 2154
configuration, 620
copying, 497
debugger configuration, 1880
.ini, 620
large files, 2197
mime.types, 1337
modes, 23
option de ligne de commande compileall, 2156
option de ligne de commande gzip, 574
path configuration, 2050
.pdbrc, 1880
plist, 642
temporary, 487
--file
option de ligne de commande trace, 1899
file (*attribut bdb.Breakpoint*), 1870
file (*attribut pyclbr.Class*), 2153
file (*attribut pyclbr.Function*), 2153

file control
    UNIX, 2204
file name
    temporary, 487
file object
    io module, 730
    open() built-in function, 23
FILE_ATTRIBUTE_ARCHIVE (*dans le module stat*), 484
FILE_ATTRIBUTE_COMPRESSED (*dans le module stat*), 484
FILE_ATTRIBUTE_DEVICE (*dans le module stat*), 484
FILE_ATTRIBUTE_DIRECTORY (*dans le module stat*), 484
FILE_ATTRIBUTE_ENCRYPTED (*dans le module stat*), 484
FILE_ATTRIBUTE_HIDDEN (*dans le module stat*), 484
FILE_ATTRIBUTE_INTEGRITY_STREAM (*dans le module stat*), 484
FILE_ATTRIBUTE_NO_SCRUB_DATA (*dans le module stat*), 484
FILE_ATTRIBUTE_NORMAL (*dans le module stat*), 484
FILE_ATTRIBUTE_NOT_CONTENT_INDEXED (*dans le module stat*), 484
FILE_ATTRIBUTE_OFFLINE (*dans le module stat*), 484
FILE_ATTRIBUTE_READONLY (*dans le module stat*), 484
FILE_ATTRIBUTE_REPARSE_POINT (*dans le module stat*), 484
FILE_ATTRIBUTE_SPARSE_FILE (*dans le module stat*), 484
FILE_ATTRIBUTE_SYSTEM (*dans le module stat*), 484
FILE_ATTRIBUTE_TEMPORARY (*dans le module stat*), 484
FILE_ATTRIBUTE_VIRTUAL (*dans le module stat*), 484
file_digest() (*dans le module hashlib*), 648
file_open() (*méthode urllib.request.FileHandler*), 1437
file_size (*attribut zipfile.ZipInfo*), 594
filecmp
    module, 485
fileConfig() (*dans le module logging.config*), 775
FileCookieJar (*classe dans http.cookiejar*), 1513
FileDialog (*classe dans tkinter.filedialog*), 1636
FileExistsError, 115
FileFinder (*classe dans importlib.machinery*), 2076
FileHandler (*classe dans logging*), 787
FileHandler (*classe dans urllib.request*), 1431
fileinput
    module, 921
FileInput (*classe dans fileinput*), 922
FileIO (*classe dans io*), 737
filelineno() (*dans le module fileinput*), 922
FileLoader (*classe dans importlib.abc*), 2071
filemode() (*dans le module stat*), 480
filename (*attribut doctest.DocTest*), 1743
filename (*attribut http.cookiejar.FileCookieJar*), 1515
filename (*attribut inspect.FrameInfo*), 2039
filename (*attribut inspect.Traceback*), 2039
filename (*attribut netrc.NetrcParseError*), 641
filename (*attribut OSError*), 111
filename (*attribut SyntaxError*), 112
filename (*attribut traceback.FrameSummary*), 2016
filename (*attribut traceback.TracebackException*), 2014
filename (*attribut tracemalloc.Frame*), 1908
filename (*attribut zipfile.ZipFile*), 590
filename (*attribut zipfile.ZipInfo*), 593
filename() (*dans le module fileinput*), 922
filename2 (*attribut OSError*), 111
filename_only (*dans le module tabnanny*), 2152
filename_pattern (*attribut tracemalloc.Filter*), 1908
filenames
    pathname expansion, 492
    wildcard expansion, 495
fileno() (*dans le module fileinput*), 922
fileno() (*méthode bz2.BZ2File*), 575
fileno() (*méthode http.client.HTTPResponse*), 1462
fileno() (*méthode io.IOBase*), 734
fileno() (*méthode multiprocessing.connection.Connection*), 984
fileno() (*méthode select.devpoll*), 1226
fileno() (*méthode select.epoll*), 1227
fileno() (*méthode select.kqueue*), 1229
fileno() (*méthode selectors.DevpollSelector*), 1233
fileno() (*méthode selectors.EpollSelector*), 1233
fileno() (*méthode selectors.KqueueSelector*), 1233
fileno() (*méthode socketserver.BaseServer*), 1496
fileno() (*méthode socket.socket*), 1179
FileNotFoundError, 115
fileobj (*attribut selectors.SelectorKey*), 1232
files() (*dans le module importlib.metadata*), 2093
files() (*dans le module importlib.resources*), 2085
files() (*méthode importlib.abc.TraversableResources*), 2074
files() (*méthode importlib.resources.abc.TraversableResources*), 2090
files_double_event() (*méthode tkinter.filedialog.FileDialog*), 1636
files_select_event() (*méthode tkinter.filedialog.FileDialog*), 1636
FileType (*classe dans argparse*), 874
FileWrapper (*classe dans wsgiref.types*), 1424
FileWrapper (*classe dans wsgiref.util*), 1417
fill() (*dans le module textwrap*), 168
fill() (*dans le module turtle*), 1588
fill() (*méthode textwrap.TextWrapper*), 172
fillcolor() (*dans le module turtle*), 1587
filling() (*dans le module turtle*), 1588
fillvalue (*attribut reprlib.Repr*), 314
--filter
    option de ligne de commande tarfile, 609
filter (*attribut select.kevent*), 1229
Filter (*classe dans logging*), 765
Filter (*classe dans tracemalloc*), 1907

filter()
built-in function, 16
filter() (*dans le module curses*), 925
filter() (*dans le module fnmatch*), 496
filter() (*méthode logging.Filter*), 765
filter() (*méthode logging.Handler*), 763
filter() (*méthode logging.Logger*), 761
filter_command() (*méthode tkinter.filedialog.FileDialog*), 1636
FILTER_DIR (*dans le module unittest.mock*), 1818
filter_traces() (*méthode tracemalloc.Snapshot*), 1908
FilterError, 597
filterfalse() (*dans le module itertools*), 414
filterwarnings() (*dans le module warnings*), 1977
Final (*dans le module typing*), 1689
final() (*dans le module typing*), 1714
finalize (*classe dans weakref*), 295
find() (*dans le module gettext*), 1554
find() (*méthode bytearray*), 69
find() (*méthode bytes*), 69
find() (*méthode doctest.DocTestFinder*), 1745
find() (*méthode mmap.mmap*), 1246
find() (*méthode str*), 55
find() (*méthode xml.etree.ElementTree.Element*), 1367
find() (*méthode xml.etree.ElementTree.ElementTree*), 1369
find_class() (*méthode pickle.Unpickler*), 514
find_class() (*pickle protocol*), 523
find_library() (*dans le module ctypes.util*), 840
find_longest_match() (*méthode difflib.SequenceMatcher*), 161
find_msvcrt() (*dans le module ctypes.util*), 840
find_spec() (*dans le module importlib.util*), 2080
find_spec() (*méthode de la classe importlib.machinery.PathFinder*), 2075
find_spec() (*méthode importlib.abc.MetaPathFinder*), 2068
find_spec() (*méthode importlib.abc.PathEntryFinder*), 2069
find_spec() (*méthode importlib.machinery.FileFinder*), 2076
find_spec() (*méthode zipimport.zipimporter*), 2058
find_unused_port() (*dans le module test.support.socket_helper*), 1856
find_user_password() (*méthode urllib.request.HTTPPasswordMgr*), 1436
find_user_password() (*méthode urllib.request.HTTPPasswordMgrWithPriorAuth*), 1436
findall() (*dans le module re*), 143
findall() (*méthode re.Pattern*), 147
findall() (*méthode xml.etree.ElementTree.Element*), 1367
findall() (*méthode xml.etree.ElementTree.ElementTree*), 1369
findCaller() (*méthode logging.Logger*), 761
findfile() (*dans le module test.support*), 1850
finditer() (*dans le module re*), 144
finditer() (*méthode re.Pattern*), 147
findlabels() (*dans le module dis*), 2164
findlinestarts() (*dans le module dis*), 2164
findtext() (*méthode xml.etree.ElementTree.Element*), 1367
findtext() (*méthode xml.etree.ElementTree.ElementTree*), 1369
finish() (*méthode socketserver.BaseRequestHandler*), 1498
finish() (*méthode tkinter.dnd.DndHandler*), 1640
finish_request() (*méthode socketserver.BaseServer*), 1497
FIRST_COMPLETED (*dans le module asyncio*), 1077
FIRST_COMPLETED (*dans le module concurrent.futures*), 1027
FIRST_EXCEPTION (*dans le module asyncio*), 1077
FIRST_EXCEPTION (*dans le module concurrent.futures*), 1027
firstChild (*attribut xml.dom.Node*), 1377
firstkey() (*méthode dbm.gnu.gdbm*), 533
--first-weekday
option de ligne de commande calendar, 258
firstweekday (*attribut calendar.Calendar*), 250
firstweekday() (*dans le module calendar*), 254
fix_missing_locations() (*dans le module ast*), 2135
fix_sentence_endings (*attribut textwrap.TextWrapper*), 171
Flag (*classe dans enum*), 323
flag_bits (*attribut zipfile.ZipInfo*), 594
FlagBoundary (*classe dans enum*), 327
flags (*attribut re.Pattern*), 147
flags (*attribut select.kevent*), 1229
flags (*dans le module sys*), 1935
flash() (*dans le module curses*), 925
flatten() (*méthode email.generator.BytesGenerator*), 1262
flatten() (*méthode email.generator.Generator*), 1263
flattening
objects, 509
float
built-in function, 39
--float
option de ligne de commande random, 392
float (*classe de base*), 16
float_info (*dans le module sys*), 1937
float_repr_style (*dans le module sys*), 1939
floating-point
literals, 39
object, 38
FloatingPointError, 109
FloatOperation (*classe dans decimal*), 371
flock() (*dans le module fcntl*), 2205
floor() (*dans le module math*), 341
floor() (*in module math*), 40

FloorDiv (*classe dans ast*), 2110
floordiv() (*dans le module operator*), 438
flush() (*méthode bz2.BZ2Compressor*), 576
flush() (*méthode io.BufferedWriter*), 739
flush() (*méthode io.IOBase*), 734
flush() (*méthode logging.Handler*), 763
flush() (*méthode logging.handlers.BufferingHandler*), 796
flush() (*méthode logging.handlers.MemoryHandler*), 797
flush() (*méthode logging.StreamHandler*), 786
flush() (*méthode lzma.LZMACompressor*), 581
flush() (*méthode mailbox.Mailbox*), 1320
flush() (*méthode mailbox.Maildir*), 1322
flush() (*méthode mailbox.MH*), 1325
flush() (*méthode mmap.mmap*), 1246
flush() (*méthode xml.etree.ElementTree.XMLParser*), 1372
flush() (*méthode xml.etree.ElementTree.XMLPullParser*), 1373
flush() (*méthode zlib.Compress*), 569
flush() (*méthode zlib.Decompress*), 570
flush_headers() (*méthode http.server.BaseHTTPRequestHandler*), 1505
flush_std_streams() (*dans le module test.support*), 1851
flushinp() (*dans le module curses*), 925
FlushKey() (*dans le module winreg*), 2188
fma() (*dans le module math*), 341
fma() (*méthode decimal.Context*), 367
fma() (*méthode decimal.Decimal*), 359
fmean() (*dans le module statistics*), 395
fmod() (*dans le module math*), 341
FMT_BINARY (*dans le module plistlib*), 644
FMT_XML (*dans le module plistlib*), 643
fnmatch
module, 495
fnmatch() (*dans le module fnmatch*), 495
fnmatchcase() (*dans le module fnmatch*), 496
focus() (*méthode tkinter.ttk.Treeview*), 1652
fold (*attribut datetime.datetime*), 222
fold (*attribut datetime.time*), 230
fold() (*méthode email.headerregistry.BaseHeader*), 1274
fold() (*méthode email.policy.Compat32*), 1271
fold() (*méthode email.policy.EmailPolicy*), 1270
fold() (*méthode email.policy.Policy*), 1268
fold_binary() (*méthode email.policy.Compat32*), 1271
fold_binary() (*méthode email.policy.EmailPolicy*), 1270
fold_binary() (*méthode email.policy.Policy*), 1269
fonction, **2235**
fonction clé, **2237**
fonction coroutine, **2232**
fonction de rappel (*callback*), **2231**
fonction générique, **2235**
Font (*classe dans tkinter.font*), 1633
For (*classe dans ast*), 2119
FOR_ITER (*opcode*), 2174
forget() (*dans le module test.support.import_helper*), 1861
forget() (*méthode tkinter.ttk.Notebook*), 1647
fork() (*dans le module os*), 715
fork() (*dans le module pty*), 2202
ForkingMixIn (*classe dans socketserver*), 1495
ForkingTCPServer (*classe dans socketserver*), 1495
ForkingUDPServer (*classe dans socketserver*), 1495
ForkingUnixDatagramServer (*classe dans socketserver*), 1495
ForkingUnixStreamServer (*classe dans socketserver*), 1495
forkpty() (*dans le module os*), 715
FORMAT (*attribut inspect.BufferFlags*), 2044
format (*attribut memoryview*), 87
format (*attribut multiprocessing.shared_memory.ShareableList*), 1018
format (*attribut struct.Struct*), 188
Format (*classe dans annotationlib*), 2046
format()
built-in function, 17
format() (*méthode inspect.Signature*), 2033
format() (*méthode logging.BufferingFormatter*), 765
format() (*méthode logging.Formatter*), 764
format() (*méthode logging.Handler*), 763
format() (*méthode pprint.PrettyPrinter*), 309
format() (*méthode str*), 55
format() (*méthode string.Formatter*), 122
format() (*méthode traceback.StackSummary*), 2015
format() (*méthode traceback.TracebackException*), 2014
format() (*méthode tracemalloc.Traceback*), 1910
format_call_graph() (*dans le module asyncio*), 1106
format_datetime() (*dans le module email.utils*), 1306
format_exc() (*dans le module traceback*), 2012
format_exception() (*dans le module traceback*), 2012
format_exception_only() (*dans le module traceback*), 2012
format_exception_only() (*méthode traceback.TracebackException*), 2014
format_field() (*méthode string.Formatter*), 123
format_frame_summary() (*méthode traceback.StackSummary*), 2015
format_help() (*méthode argparse.ArgumentParser*), 878
format_list() (*dans le module traceback*), 2012
format_map() (*méthode str*), 56
FORMAT_SIMPLE (*opcode*), 2177
format_stack() (*dans le module traceback*), 2012
format_stack_entry() (*méthode bdb.Bdb*), 1873
format_string() (*dans le module locale*), 1566
format_tb() (*dans le module traceback*), 2012

format_usage() (*méthode argparse.Action*), 868
format_usage() (*méthode argparse.ArgumentParser*), 878
FORMAT_WITH_SPEC (*opcode*), 2178
formataddr() (*dans le module email.utils*), 1304
formatargvalues() (*dans le module inspect*), 2037
formatdate() (*dans le module email.utils*), 1305
formatday() (*méthode calendar.TextCalendar*), 252
FormatError, 1335
FormatError() (*dans le module ctypes*), 840
formatException() (*méthode logging.Formatter*), 765
formatFooter() (*méthode logging.BufferingFormatter*), 765
formatHeader() (*méthode logging.BufferingFormatter*), 765
formatmonth() (*méthode calendar.HTMLCalendar*), 252
formatmonth() (*méthode calendar.TextCalendar*), 252
formatmonthname() (*méthode calendar.HTMLCalendar*), 252
formatmonthname() (*méthode calendar.TextCalendar*), 252
formatStack() (*méthode logging.Formatter*), 765
FormattedValue (*classe dans ast*), 2107
Formatter (*classe dans logging*), 764
Formatter (*classe dans string*), 122
formatTime() (*méthode logging.Formatter*), 764
formatting
    bytearray (%), 79
    bytes (%), 79
formatting, string (%), 63
formatwarning() (*dans le module warnings*), 1977
formatweek() (*méthode calendar.TextCalendar*), 252
formatweekday() (*méthode calendar.TextCalendar*), 252
formatweekheader() (*méthode calendar.TextCalendar*), 252
formatyear() (*méthode calendar.HTMLCalendar*), 252
formatyear() (*méthode calendar.TextCalendar*), 252
formatyearpage() (*méthode calendar.HTMLCalendar*), 252
Fortran contiguous, 2232
forward() (*dans le module turtle*), 1578
FORWARDREF (*attribut annotationlib.Format*), 2046
ForwardRef (*classe dans annotationlib*), 2046
ForwardRef (*classe dans typing*), 1717
fp (*attribut urllib.error.HTTPError*), 1452
fpathconf() (*dans le module os*), 675
Fraction (*classe dans fractions*), 380
fractions
    module, 380
frame (*attribut inspect.FrameInfo*), 2039
frame (*attribut tkinter.scrolledtext.ScrolledText*), 1639
Frame (*classe dans tracemalloc*), 1908
FrameInfo (*classe dans inspect*), 2039
FrameSummary (*classe dans traceback*), 2015
FrameType (*dans le module types*), 303
free threading, **2234**
free variable, **2234**
free_tool_id() (*dans le module sys.monitoring*), 1955
freedesktop_os_release() (*dans le module platform*), 803
freeze() (*dans le module gc*), 2024
freeze_support() (*dans le module multiprocessing*), 983
frexp() (*dans le module math*), 341
FRIDAY (*dans le module calendar*), 255
from_address() (*méthode ctypes._CData*), 842
from_buffer() (*méthode ctypes._CData*), 842
from_buffer_copy() (*méthode ctypes._CData*), 842
from_bytes() (*méthode de la classe int*), 42
from_callable() (*méthode de la classe inspect.Signature*), 2033
from_decimal() (*méthode de la classe fractions.Fraction*), 381
from_exception() (*méthode de la classe traceback.TracebackException*), 2014
from_file() (*méthode de la classe zipfile.ZipInfo*), 593
from_file() (*méthode de la classe zoneinfo.ZoneInfo*), 247
from_float() (*méthode de la classe decimal.Decimal*), 358
from_float() (*méthode de la classe fractions.Fraction*), 381
from_iterable() (*méthode de la classe itertools.chain*), 412
from_list() (*méthode de la classe traceback.StackSummary*), 2015
from_number() (*méthode de la classe complex*), 44
from_number() (*méthode de la classe decimal.Decimal*), 359
from_number() (*méthode de la classe float*), 43
from_number() (*méthode de la classe fractions.Fraction*), 382
from_param() (*méthode ctypes._CData*), 842
from_samples() (*méthode de la classe statistics.NormalDist*), 405
from_traceback() (*méthode de la classe dis.Bytecode*), 2161
from_uri() (*méthode de la classe pathlib.Path*), 458
frombuf() (*méthode de la classe tarfile.TarInfo*), 603
frombytes() (*méthode array.array*), 290
fromfd() (*dans le module socket*), 1172
fromfd() (*méthode select.epoll*), 1227
fromfd() (*méthode select.kqueue*), 1229
fromfile() (*méthode array.array*), 290
fromhex() (*méthode de la classe bytearray*), 66
fromhex() (*méthode de la classe bytes*), 65
fromhex() (*méthode de la classe float*), 44
fromisocalendar() (*méthode de la classe datetime.date*), 213
fromisocalendar() (*méthode de la classe datetime.datetime*), 221

fromisoformat() (*méthode de la classe datetime.date*), 213
fromisoformat() (*méthode de la classe datetime.datetime*), 220
fromisoformat() (*méthode de la classe datetime.time*), 230
fromkeys() (*méthode collections.Counter*), 263
fromkeys() (*méthode de la classe dict*), 92
fromlist() (*méthode array.array*), 291
fromordinal() (*méthode de la classe datetime.date*), 213
fromordinal() (*méthode de la classe datetime.datetime*), 220
fromshare() (*dans le module socket*), 1173
fromstring() (*dans le module xml.etree.ElementTree*), 1362
fromstringlist() (*dans le module xml.etree.ElementTree*), 1362
fromtarfile() (*méthode de la classe tarfile.TarInfo*), 603
fromtimestamp() (*méthode de la classe datetime.date*), 213
fromtimestamp() (*méthode de la classe datetime.datetime*), 219
fromunicode() (*méthode array.array*), 291
fromutc() (*méthode datetime.timezone*), 240
fromutc() (*méthode datetime.tzinfo*), 234
FrozenImporter (*classe dans importlib.machinery*), 2075
FrozenInstanceError, 1985
FrozenSet (*classe dans typing*), 1719
frozenset (*classe de base*), 88
FS (*dans le module curses.ascii*), 951
fs_is_case_insensitive() (*dans le module test.support.os_helper*), 1860
FS_NONASCII (*dans le module test.support.os_helper*), 1859
fsdecode() (*dans le module os*), 666
fsencode() (*dans le module os*), 666
fspath() (*dans le module os*), 666
fstat() (*dans le module os*), 675
fstatvfs() (*dans le module os*), 675
FSTRING_END (*dans le module token*), 2146
FSTRING_MIDDLE (*dans le module token*), 2146
FSTRING_START (*dans le module token*), 2146
fsum() (*dans le module math*), 344
fsync() (*dans le module os*), 675
FTP, 1442
    ftplib (*standard module*), 1465
    protocol, 1442, 1465
FTP (*classe dans ftplib*), 1465
ftp_open() (*méthode urllib.request.FTPHandler*), 1438
FTP_TLS (*classe dans ftplib*), 1469
FTPHandler (*classe dans urllib.request*), 1431
ftplib
    module, 1465
ftruncate() (*dans le module os*), 675
Full, 1049
FULL (*attribut inspect.BufferFlags*), 2044
full() (*méthode asyncio.Queue*), 1102
full() (*méthode multiprocessing.Queue*), 980
full() (*méthode queue.Queue*), 1050
full_match() (*méthode pathlib.PurePath*), 454
FULL_RO (*attribut inspect.BufferFlags*), 2044
full_url (*attribut urllib.request.Request*), 1431
fullmatch() (*dans le module re*), 143
fullmatch() (*méthode re.Pattern*), 147
fully_trusted_filter() (*dans le module tarfile*), 606
func (*attribut functools.partial*), 437
funcname (*attribut bdb.Breakpoint*), 1870
function (*attribut inspect.FrameInfo*), 2039
function (*attribut inspect.Traceback*), 2039
FUNCTION (*attribut symtable.SymbolTableType*), 2139
Function (*classe dans pyclbr*), 2153
Function (*classe dans symtable*), 2141
FunctionDef (*classe dans ast*), 2130
FunctionTestCase (*classe dans unittest*), 1773
FunctionType (*classe dans ast*), 2106
FunctionType (*dans le module types*), 301
functools
    module, 426
funny_files (*attribut filecmp.dircmp*), 486
Future (*classe dans asyncio*), 1131
Future (*classe dans concurrent.futures*), 1025
future_add_to_awaited_by() (*dans le module asyncio*), 1107
future_discard_from_awaited_by() (*dans le module asyncio*), 1107
FutureWarning, 116
fwalk() (*dans le module os*), 705

G

-g
    option de ligne de commande trace, 1900
gaierror, 1165
gamma() (*dans le module math*), 346
gammavariate() (*dans le module random*), 387
garbage (*dans le module gc*), 2024
gather() (*dans le module asyncio*), 1071
gather() (*méthode curses.textpad.Textbox*), 949
gauss() (*dans le module random*), 387
gc
    module, 2021
gc_collect() (*dans le module test.support*), 1851
gcd() (*dans le module math*), 340
ge() (*dans le module operator*), 437
generate_tokens() (*dans le module tokenize*), 2148
générateur, **2235**
générateur asynchrone, **2230**
Generator (*classe dans collections.abc*), 279
Generator (*classe dans email.generator*), 1263
Generator (*classe dans typing*), 1723
GeneratorExit, 109

GeneratorExp (*classe dans ast*), 2113
GeneratorType (*dans le module types*), 301
Generic
    Alias, 96
Generic (*classe dans typing*), 1694
generic_visit() (*méthode ast.NodeVisitor*), 2136
GenericAlias
    object, 96
GenericAlias (*classe dans types*), 303
genops() (*dans le module pickletools*), 2182
geometric_mean() (*dans le module statistics*), 396
gestionnaire de contexte, **2232**
gestionnaire de contexte asynchrone, **2230**
get() (*dans le module webbrowser*), 1414
get() (*méthode asyncio.Queue*), 1102
get() (*méthode configparser.ConfigParser*), 636
get() (*méthode contextvars.Context*), 1055
get() (*méthode contextvars.ContextVar*), 1053
get() (*méthode dict*), 92
get() (*méthode email.message.EmailMessage*), 1253
get() (*méthode email.message.Message*), 1291
get() (*méthode mailbox.Mailbox*), 1319
get() (*méthode multiprocessing.pool.AsyncResult*), 1000
get() (*méthode multiprocessing.Queue*), 981
get() (*méthode multiprocessing.SimpleQueue*), 982
get() (*méthode queue.Queue*), 1050
get() (*méthode queue.SimpleQueue*), 1052
get() (*méthode tkinter.ttk.Combobox*), 1645
get() (*méthode tkinter.ttk.Spinbox*), 1646
get() (*méthode types.MappingProxyType*), 304
get() (*méthode xml.etree.ElementTree.Element*), 1366
GET_AITER (*opcode*), 2168
get_all() (*méthode email.message.EmailMessage*), 1253
get_all() (*méthode email.message.Message*), 1292
get_all() (*méthode wsgiref.headers.Headers*), 1418
get_all_breaks() (*méthode bdb.Bdb*), 1873
get_all_start_methods() (*dans le module multiprocessing*), 983
GET_ANEXT (*opcode*), 2168
get_annotate_function() (*dans le module annotationlib*), 2048
get_annotations() (*dans le module annotationlib*), 2048
get_annotations() (*dans le module inspect*), 2038
get_app() (*méthode wsgiref.simple_server.WSGIServer*), 1419
get_archive_formats() (*dans le module shutil*), 504
get_args() (*dans le module typing*), 1716
get_asyncgen_hooks() (*dans le module sys*), 1942
get_attribute() (*dans le module test.support*), 1853
GET_AWAITABLE (*opcode*), 2168
get_begidx() (*dans le module readline*), 178
get_blocking() (*dans le module os*), 675
get_body() (*méthode email.message.EmailMessage*), 1256
get_body_encoding() (*méthode email.charset.Charset*), 1302
get_boundary() (*méthode email.message.EmailMessage*), 1254
get_boundary() (*méthode email.message.Message*), 1294
get_bpbynumber() (*méthode bdb.Bdb*), 1872
get_break() (*méthode bdb.Bdb*), 1873
get_breaks() (*méthode bdb.Bdb*), 1873
get_buffer() (*méthode asyncio.BufferedProtocol*), 1141
get_bytes() (*méthode mailbox.Mailbox*), 1319
get_ca_certs() (*méthode ssl.SSLContext*), 1208
get_cache_token() (*dans le module abc*), 2008
get_channel_binding() (*méthode ssl.SSLSocket*), 1204
get_charset() (*méthode email.message.Message*), 1290
get_charsets() (*méthode email.message.EmailMessage*), 1255
get_charsets() (*méthode email.message.Message*), 1294
get_children() (*méthode symtable.SymbolTable*), 2141
get_children() (*méthode tkinter.ttk.Treeview*), 1651
get_ciphers() (*méthode ssl.SSLContext*), 1208
get_clock_info() (*dans le module time*), 746
get_close_matches() (*dans le module difflib*), 158
get_code() (*méthode importlib.abc.InspectLoader*), 2070
get_code() (*méthode importlib.abc.SourceLoader*), 2072
get_code() (*méthode importlib.machinery.ExtensionFileLoader*), 2077
get_code() (*méthode importlib.machinery.SourcelessFileLoader*), 2077
get_code() (*méthode zipimport.zipimporter*), 2058
get_completer() (*dans le module readline*), 178
get_completer_delims() (*dans le module readline*), 178
get_completion_type() (*dans le module readline*), 178
get_config_h_filename() (*dans le module sysconfig*), 1964
get_config_var() (*dans le module sysconfig*), 1960
get_config_vars() (*dans le module sysconfig*), 1960
get_content() (*dans le module email.contentmanager*), 1280
get_content() (*méthode email.contentmanager.ContentManager*), 1279
get_content() (*méthode email.message.EmailMessage*), 1257
get_content_charset() (*méthode email.message.EmailMessage*), 1255
get_content_charset() (*méthode email.message.Message*), 1294
get_content_disposition() (*méthode email.message.EmailMessage*), 1255
get_content_disposition() (*méthode*

email.message.Message), 1294
get_content_maintype() (*méthode email.message.EmailMessage*), 1254
get_content_maintype() (*méthode email.message.Message*), 1292
get_content_subtype() (*méthode email.message.EmailMessage*), 1254
get_content_subtype() (*méthode email.message.Message*), 1292
get_content_type() (*méthode email.message.EmailMessage*), 1253
get_content_type() (*méthode email.message.Message*), 1292
get_context() (*dans le module multiprocessing*), 983
get_context() (*méthode asyncio.Handle*), 1125
get_context() (*méthode asyncio.Task*), 1082
get_coro() (*méthode asyncio.Task*), 1082
get_coroutine_origin_tracking_depth() (*dans le module sys*), 1942
get_count() (*dans le module gc*), 2023
get_current_history_length() (*dans le module readline*), 177
get_data() (*dans le module pkgutil*), 2061
get_data() (*méthode importlib.abc.FileLoader*), 2071
get_data() (*méthode importlib.abc.ResourceLoader*), 2070
get_data() (*méthode zipimport.zipimporter*), 2058
get_date() (*méthode mailbox.MaildirMessage*), 1328
get_debug() (*dans le module gc*), 2022
get_debug() (*méthode asyncio.loop*), 1124
get_default() (*méthode argparse.ArgumentParser*), 877
get_default_scheme() (*dans le module sysconfig*), 1963
get_default_type() (*méthode email.message.EmailMessage*), 1254
get_default_type() (*méthode email.message.Message*), 1293
get_default_verify_paths() (*dans le module ssl*), 1194
get_dialect() (*dans le module csv*), 614
get_disassembly_as_string() (*méthode test.support.bytecode_helper.BytecodeTestCase*), 1858
get_docstring() (*dans le module ast*), 2135
get_doctest() (*méthode doctest.DocTestParser*), 1745
get_endidx() (*dans le module readline*), 178
get_environ() (*méthode wsgiref.simple_server.WSGIRequestHandler*), 1420
get_errno() (*dans le module ctypes*), 840
get_escdelay() (*dans le module curses*), 929
get_event_loop() (*dans le module asyncio*), 1107
get_event_loop() (*méthode asyncio.AbstractEventLoopPolicy*), 1148
get_event_loop_policy() (*dans le module asyncio*), 1148
get_events() (*dans le module sys.monitoring*), 1958
get_examples() (*méthode doctest.DocTestParser*), 1745
get_exception_handler() (*méthode asyncio.loop*), 1123
get_exec_path() (*dans le module os*), 667
get_extra_info() (*méthode asyncio.BaseTransport*), 1136
get_extra_info() (*méthode asyncio.StreamWriter*), 1088
get_field() (*méthode string.Formatter*), 122
get_file() (*méthode mailbox.Babyl*), 1325
get_file() (*méthode mailbox.Mailbox*), 1319
get_file() (*méthode mailbox.Maildir*), 1323
get_file() (*méthode mailbox.mbox*), 1323
get_file() (*méthode mailbox.MH*), 1325
get_file() (*méthode mailbox.MMDF*), 1326
get_file_breaks() (*méthode bdb.Bdb*), 1873
get_filename() (*méthode email.message.EmailMessage*), 1254
get_filename() (*méthode email.message.Message*), 1294
get_filename() (*méthode importlib.abc.ExecutionLoader*), 2071
get_filename() (*méthode importlib.abc.FileLoader*), 2071
get_filename() (*méthode importlib.machinery.ExtensionFileLoader*), 2078
get_filename() (*méthode zipimport.zipimporter*), 2058
get_filter() (*méthode tkinter.filedialog.FileDialog*), 1636
get_flags() (*méthode mailbox.Maildir*), 1321
get_flags() (*méthode mailbox.MaildirMessage*), 1327
get_flags() (*méthode mailbox.mboxMessage*), 1329
get_flags() (*méthode mailbox.MMDFMessage*), 1333
get_folder() (*méthode mailbox.Maildir*), 1321
get_folder() (*méthode mailbox.MH*), 1324
get_frees() (*méthode symtable.Function*), 2141
get_freeze_count() (*dans le module gc*), 2024
get_from() (*méthode mailbox.mboxMessage*), 1329
get_from() (*méthode mailbox.MMDFMessage*), 1333
get_full_url() (*méthode urllib.request.Request*), 1432
get_globals() (*méthode symtable.Function*), 2141
get_grouped_opcodes() (*méthode difflib.SequenceMatcher*), 162
get_handle_inheritable() (*dans le module os*), 685
get_header() (*méthode urllib.request.Request*), 1432
get_history_item() (*dans le module readline*), 177
get_history_length() (*dans le module readline*), 177
get_id() (*méthode symtable.SymbolTable*), 2140
get_ident() (*dans le module _thread*), 1057
get_ident() (*dans le module threading*), 956
get_identifiers() (*méthode string.Template*), 131

get_identifiers() (*méthode symtable.SymbolTable*), 2140
get_importer() (*dans le module pkgutil*), 2060
get_info() (*méthode mailbox.Maildir*), 1322
get_info() (*méthode mailbox.MaildirMessage*), 1328
get_inheritable() (*dans le module os*), 685
get_inheritable() (*méthode socket.socket*), 1180
get_instructions() (*dans le module dis*), 2164
get_int_max_str_digits() (*dans le module sys*), 1940
get_interpreter() (*dans le module zipapp*), 1926
GET_ITER (*opcode*), 2167
get_key() (*méthode selectors.BaseSelector*), 1233
get_labels() (*méthode mailbox.Babyl*), 1325
get_labels() (*méthode mailbox.BabylMessage*), 1332
get_last_error() (*dans le module ctypes*), 840
GET_LEN (*opcode*), 2170
get_line_buffer() (*dans le module readline*), 176
get_lineno() (*méthode symtable.SymbolTable*), 2140
get_local_events() (*dans le module sys.monitoring*), 1958
get_locals() (*méthode symtable.Function*), 2141
get_logger() (*dans le module multiprocessing*), 1004
get_loop() (*méthode asyncio.Future*), 1133
get_loop() (*méthode asyncio.Runner*), 1064
get_loop() (*méthode asyncio.Server*), 1126
get_makefile_filename() (*dans le module sysconfig*), 1964
get_map() (*méthode selectors.BaseSelector*), 1233
get_matching_blocks() (*méthode difflib.SequenceMatcher*), 161
get_message() (*méthode mailbox.Mailbox*), 1319
get_method() (*méthode urllib.request.Request*), 1432
get_methods() (*méthode symtable.Class*), 2141
get_mixed_type_key() (*dans le module ipaddress*), 1547
get_name() (*méthode asyncio.Task*), 1082
get_name() (*méthode symtable.Symbol*), 2141
get_name() (*méthode symtable.SymbolTable*), 2140
get_namespace() (*méthode symtable.Symbol*), 2143
get_namespaces() (*méthode symtable.Symbol*), 2143
get_native_id() (*dans le module _thread*), 1057
get_native_id() (*dans le module threading*), 956
get_nonlocals() (*méthode symtable.Function*), 2141
get_nonstandard_attr() (*méthode http.cookiejar.Cookie*), 1520
get_nowait() (*méthode asyncio.Queue*), 1103
get_nowait() (*méthode multiprocessing.Queue*), 981
get_nowait() (*méthode queue.Queue*), 1050
get_nowait() (*méthode queue.SimpleQueue*), 1052
get_object_traceback() (*dans le module tracemalloc*), 1906
get_objects() (*dans le module gc*), 2022
get_opcodes() (*méthode difflib.SequenceMatcher*), 161
get_option() (*méthode optparse.OptionParser*), 912
get_option_group() (*méthode optparse.OptionParser*), 903
get_origin() (*dans le module typing*), 1716
get_original_bases() (*dans le module types*), 300
get_original_stdout() (*dans le module test.support*), 1850
get_osfhandle() (*dans le module msvcrt*), 2184
get_output_charset() (*méthode email.charset.Charset*), 1302
get_overloads() (*dans le module typing*), 1713
get_pagesize() (*dans le module test.support*), 1850
get_param() (*méthode email.message.Message*), 1293
get_parameters() (*méthode symtable.Function*), 2141
get_params() (*méthode email.message.Message*), 1293
get_path() (*dans le module sysconfig*), 1963
get_path_names() (*dans le module sysconfig*), 1963
get_paths() (*dans le module sysconfig*), 1963
get_payload() (*méthode email.message.Message*), 1290
get_pid() (*méthode asyncio.SubprocessTransport*), 1138
get_pipe_transport() (*méthode asyncio.SubprocessTransport*), 1138
get_platform() (*dans le module sysconfig*), 1964
get_poly() (*dans le module turtle*), 1593
get_preferred_scheme() (*dans le module sysconfig*), 1963
get_protocol() (*méthode asyncio.BaseTransport*), 1136
get_protocol_members() (*dans le module typing*), 1716
get_proxy_response_headers() (*méthode http.client.HTTPConnection*), 1461
get_python_version() (*dans le module sysconfig*), 1964
get_ready() (*méthode graphlib.TopologicalSorter*), 332
get_referents() (*dans le module gc*), 2023
get_referrers() (*dans le module gc*), 2023
get_request() (*méthode socketserver.BaseServer*), 1497
get_returncode() (*méthode asyncio.SubprocessTransport*), 1138
get_running_loop() (*dans le module asyncio*), 1107
get_scheme() (*méthode wsgiref.handlers.BaseHandler*), 1423
get_scheme_names() (*dans le module sysconfig*), 1963
get_selection() (*méthode tkinter.filedialog.FileDialog*), 1636
get_sequences() (*méthode mailbox.MH*), 1324
get_sequences() (*méthode mailbox.MHMessage*), 1330
get_server() (*méthode multiprocessing.managers.BaseManager*), 992
get_server_certificate() (*dans le module ssl*), 1194
get_shapepoly() (*dans le module turtle*), 1592
get_source() (*méthode importlib.abc.InspectLoader*),

2070
get_source() (*méthode importlib.abc.SourceLoader*), 2072
get_source() (*méthode importlib.machinery.ExtensionFileLoader*), 2077
get_source() (*méthode importlib.machinery.SourcelessFileLoader*), 2077
get_source() (*méthode zipimport.zipimporter*), 2058
get_source_segment() (*dans le module ast*), 2135
get_stack() (*méthode asyncio.Task*), 1082
get_stack() (*méthode bdb.Bdb*), 1873
get_start_method() (*dans le module multiprocessing*), 983
get_starttag_text() (*méthode html.parser.HTMLParser*), 1349
get_stats() (*dans le module gc*), 2022
get_stats_profile() (*méthode pstats.Stats*), 1891
get_stderr() (*méthode wsgiref.handlers.BaseHandler*), 1422
get_stderr() (*méthode wsgiref.simple_server.WSGIRequestHandler*), 1420
get_stdin() (*méthode wsgiref.handlers.BaseHandler*), 1422
get_string() (*méthode mailbox.Mailbox*), 1319
get_subdir() (*méthode mailbox.MaildirMessage*), 1327
get_symbols() (*méthode symtable.SymbolTable*), 2140
get_tabsize() (*dans le module curses*), 929
get_task_factory() (*méthode asyncio.loop*), 1112
get_terminal_size() (*dans le module os*), 685
get_terminal_size() (*dans le module shutil*), 506
get_threshold() (*dans le module gc*), 2023
get_token() (*méthode shlex.shlex*), 1615
get_tool() (*dans le module sys.monitoring*), 1955
get_traceback_limit() (*dans le module tracemalloc*), 1906
get_traced_memory() (*dans le module tracemalloc*), 1906
get_tracemalloc_memory() (*dans le module tracemalloc*), 1906
get_type() (*méthode symtable.SymbolTable*), 2140
get_type_hints() (*dans le module typing*), 1715
get_unixfrom() (*méthode email.message.EmailMessage*), 1252
get_unixfrom() (*méthode email.message.Message*), 1289
get_unpack_formats() (*dans le module shutil*), 505
get_unverified_chain() (*méthode ssl.SSLSocket*), 1204
get_usage() (*méthode optparse.OptionParser*), 913
get_value() (*méthode string.Formatter*), 123
get_verified_chain() (*méthode ssl.SSLSocket*), 1204
get_version() (*méthode optparse.OptionParser*), 903
get_visible() (*méthode mailbox.BabylMessage*), 1332
get_wch() (*méthode curses.window*), 933
get_write_buffer_limits() (*méthode asyncio.WriteTransport*), 1137
get_write_buffer_size() (*méthode asyncio.WriteTransport*), 1137
GET_YIELD_FROM_ITER (*opcode*), 2167
getacl() (*méthode imaplib.IMAP4*), 1477
getaddresses() (*dans le module email.utils*), 1305
getaddrinfo() (*dans le module socket*), 1173
getaddrinfo() (*méthode asyncio.loop*), 1120
getallocatedblocks() (*dans le module sys*), 1939
getandroidapilevel() (*dans le module sys*), 1939
getannotation() (*méthode imaplib.IMAP4*), 1477
getargvalues() (*dans le module inspect*), 2037
getasyncgenlocals() (*dans le module inspect*), 2042
getasyncgenstate() (*dans le module inspect*), 2042
getatime() (*dans le module os.path*), 474
getattr()
built-in function, 18
getattr_static() (*dans le module inspect*), 2041
getAttribute() (*méthode xml.dom.Element*), 1380
getAttributeNode() (*méthode xml.dom.Element*), 1380
getAttributeNodeNS() (*méthode xml.dom.Element*), 1380
getAttributeNS() (*méthode xml.dom.Element*), 1380
GetBase() (*méthode xml.parsers.expat.xmlparser*), 1404
getbegyx() (*méthode curses.window*), 933
getbkgd() (*méthode curses.window*), 933
getblocking() (*méthode socket.socket*), 1180
getboolean() (*méthode configparser.ConfigParser*), 636
getbuffer() (*méthode io.BytesIO*), 738
getByteStream() (*méthode xml.sax.xmlreader.InputSource*), 1402
getcallargs() (*dans le module inspect*), 2037
getcanvas() (*dans le module turtle*), 1600
getch() (*dans le module msvcrt*), 2184
getch() (*méthode curses.window*), 933
getCharacterStream() (*méthode xml.sax.xmlreader.InputSource*), 1402
getche() (*dans le module msvcrt*), 2184
getChild() (*méthode logging.Logger*), 759
getChildren() (*méthode logging.Logger*), 759
getclasstree() (*dans le module inspect*), 2036
getclosurevars() (*dans le module inspect*), 2038
getcode() (*méthode http.client.HTTPResponse*), 1463
getcode() (*méthode urllib.response.addinfourl*), 1442
getColumnNumber() (*méthode xml.sax.xmlreader.Locator*), 1401
getcomments() (*dans le module inspect*), 2031
getcompname() (*méthode wave.Wave_read*), 1550
getcomptype() (*méthode wave.Wave_read*), 1550
getconfig() (*méthode sqlite3.Connection*), 550
getContentHandler() (*méthode xml.sax.xmlreader.XMLReader*), 1400
getcontext() (*dans le module decimal*), 363

getcoroutinelocals() (*dans le module inspect*), 2042
getcoroutinestate() (*dans le module inspect*), 2042
getctime() (*dans le module os.path*), 474
getcwd() (*dans le module os*), 689
getcwdb() (*dans le module os*), 689
getdecoder() (*dans le module codecs*), 190
getdefaultencoding() (*dans le module sys*), 1939
getdefaultlocale() (*dans le module locale*), 1565
getdefaulttimeout() (*dans le module socket*), 1177
getdlopenflags() (*dans le module sys*), 1939
getdoc() (*dans le module inspect*), 2030
getDOMImplementation() (*dans le module xml.dom*), 1375
getDTDHandler() (*méthode xml.sax.xmlreader.XMLReader*), 1400
getEffectiveLevel() (*méthode logging.Logger*), 759
getegid() (*dans le module os*), 667
getElementsByTagName() (*méthode xml.dom.Document*), 1379
getElementsByTagName() (*méthode xml.dom.Element*), 1380
getElementsByTagNameNS() (*méthode xml.dom.Document*), 1380
getElementsByTagNameNS() (*méthode xml.dom.Element*), 1380
getencoder() (*dans le module codecs*), 190
getencoding() (*dans le module locale*), 1566
getEncoding() (*méthode xml.sax.xmlreader.InputSource*), 1401
getEntityResolver() (*méthode xml.sax.xmlreader.XMLReader*), 1400
getenv() (*dans le module os*), 667
getenvb() (*dans le module os*), 667
getErrorHandler() (*méthode xml.sax.xmlreader.XMLReader*), 1400
geteuid() (*dans le module os*), 667
getEvent() (*méthode xml.dom.pulldom.DOMEventStream*), 1390
getEventCategory() (*méthode logging.handlers.NTEventLogHandler*), 795
getEventType() (*méthode logging.handlers.NTEventLogHandler*), 795
getException() (*méthode xml.sax.SAXException*), 1392
getFeature() (*méthode xml.sax.xmlreader.XMLReader*), 1400
getfile() (*dans le module inspect*), 2031
getFilesToDelete() (*méthode logging.handlers.TimedRotatingFileHandler*), 791
getfilesystemencodeerrors() (*dans le module sys*), 1940
getfilesystemencoding() (*dans le module sys*), 1939
getfirstweekday() (*méthode calendar.Calendar*), 250
getfloat() (*méthode configparser.ConfigParser*), 636
getfqdn() (*dans le module socket*), 1174
getframeinfo() (*dans le module inspect*), 2040
getframerate() (*méthode wave.Wave_read*), 1550
getfullargspec() (*dans le module inspect*), 2036
getgeneratorlocals() (*dans le module inspect*), 2042
getgeneratorstate() (*dans le module inspect*), 2042
getgid() (*dans le module os*), 667
getgrall() (*dans le module grp*), 2199
getgrgid() (*dans le module grp*), 2199
getgrnam() (*dans le module grp*), 2199
getgrouplist() (*dans le module os*), 667
getgroups() (*dans le module os*), 667
getHandlerByName() (*dans le module logging*), 771
getHandlerNames() (*dans le module logging*), 771
getheader() (*méthode http.client.HTTPResponse*), 1462
getheaders() (*méthode http.client.HTTPResponse*), 1462
gethostbyaddr() (*dans le module socket*), 1174
gethostbyaddr() (*in module socket*), 672
gethostbyname() (*dans le module socket*), 1174
gethostbyname_ex() (*dans le module socket*), 1174
gethostname() (*dans le module socket*), 1174
gethostname() (*in module socket*), 672
getincrementaldecoder() (*dans le module codecs*), 190
getincrementalencoder() (*dans le module codecs*), 190
getinfo() (*méthode zipfile.ZipFile*), 587
getinnerframes() (*dans le module inspect*), 2040
GetInputContext() (*méthode xml.parsers.expat.xmlparser*), 1404
getint() (*méthode configparser.ConfigParser*), 636
getitem() (*dans le module operator*), 440
getitimer() (*dans le module signal*), 1240
getkey() (*méthode curses.window*), 933
GetLastError() (*dans le module ctypes*), 840
getLength() (*méthode xml.sax.xmlreader.Attributes*), 1402
getLevelName() (*dans le module logging*), 771
getLevelNamesMapping() (*dans le module logging*), 771
getlimit() (*méthode sqlite3.Connection*), 549
getline() (*dans le module linecache*), 496
getLineNumber() (*méthode xml.sax.xmlreader.Locator*), 1401
getloadavg() (*dans le module os*), 727
getlocale() (*dans le module locale*), 1565
getLogger() (*dans le module logging*), 769
getLoggerClass() (*dans le module logging*), 769
getlogin() (*dans le module os*), 668
getLogRecordFactory() (*dans le module logging*), 770
getMandatoryRelease() (*méthode __future__._Feature*), 2021
getmark() (*méthode wave.Wave_read*), 1550
getmarkers() (*méthode wave.Wave_read*), 1550

getmaxyx() (*méthode curses.window*), 933
getmember() (*méthode tarfile.TarFile*), 600
getmembers() (*dans le module inspect*), 2027
getmembers() (*méthode tarfile.TarFile*), 600
getmembers_static() (*dans le module inspect*), 2027
getMessage() (*méthode logging.LogRecord*), 766
getMessage() (*méthode xml.sax.SAXException*), 1392
getMessageID() (*méthode logging.handlers.NTEventLogHandler*), 796
getmodule() (*dans le module inspect*), 2031
getmodulename() (*dans le module inspect*), 2028
getmouse() (*dans le module curses*), 925
getmro() (*dans le module inspect*), 2037
getmtime() (*dans le module os.path*), 474
getName() (*méthode threading.Thread*), 960
getNameByQName() (*méthode xml.sax.xmlreader.AttributesNS*), 1402
getnameinfo() (*dans le module socket*), 1175
getnameinfo() (*méthode asyncio.loop*), 1120
getnames() (*méthode tarfile.TarFile*), 600
getNames() (*méthode xml.sax.xmlreader.Attributes*), 1402
getnchannels() (*méthode wave.Wave_read*), 1550
getnframes() (*méthode wave.Wave_read*), 1550
getnode, 1491
getnode() (*dans le module uuid*), 1490
getobjects() (*dans le module sys*), 1941
getopt
    module, 2217
getopt() (*dans le module getopt*), 2217
GetoptError, 2218
getOptionalRelease() (*méthode __future__._Feature*), 2021
getouterframes() (*dans le module inspect*), 2040
getoutput() (*dans le module subprocess*), 1046
getpagesize() (*dans le module resource*), 2210
getparams() (*méthode wave.Wave_read*), 1550
getparyx() (*méthode curses.window*), 933
getpass
    module, 920
getpass() (*dans le module getpass*), 920
GetPassWarning, 921
getpeercert() (*méthode ssl.SSLSocket*), 1203
getpeername() (*méthode socket.socket*), 1180
getpen() (*dans le module turtle*), 1594
getpgid() (*dans le module os*), 668
getpgrp() (*dans le module os*), 668
getpid() (*dans le module os*), 668
getpos() (*méthode html.parser.HTMLParser*), 1349
getppid() (*dans le module os*), 668
getpreferredencoding() (*dans le module locale*), 1565
getpriority() (*dans le module os*), 668
getprofile() (*dans le module sys*), 1941
getprofile() (*dans le module threading*), 957
getProperty() (*méthode xml.sax.xmlreader.XMLReader*), 1400
getprotobyname() (*dans le module socket*), 1175
getproxies() (*dans le module urllib.request*), 1428
getPublicId() (*méthode xml.sax.xmlreader.InputSource*), 1401
getPublicId() (*méthode xml.sax.xmlreader.Locator*), 1401
getpwall() (*dans le module pwd*), 2199
getpwnam() (*dans le module pwd*), 2199
getpwuid() (*dans le module pwd*), 2198
getQNameByName() (*méthode xml.sax.xmlreader.AttributesNS*), 1402
getQNames() (*méthode xml.sax.xmlreader.AttributesNS*), 1402
getquota() (*méthode imaplib.IMAP4*), 1477
getquotaroot() (*méthode imaplib.IMAP4*), 1477
getrandbits() (*dans le module random*), 385
getrandbits() (*méthode random.Random*), 388
getrandom() (*dans le module os*), 729
getreader() (*dans le module codecs*), 190
getrecursionlimit() (*dans le module sys*), 1940
getrefcount() (*dans le module sys*), 1940
GetReparseDeferralEnabled() (*méthode xml.parsers.expat.xmlparser*), 1405
getresgid() (*dans le module os*), 669
getresponse() (*méthode http.client.HTTPConnection*), 1460
getresuid() (*dans le module os*), 669
getrlimit() (*dans le module resource*), 2207
getroot() (*méthode xml.etree.ElementTree.ElementTree*), 1369
getrusage() (*dans le module resource*), 2209
getsampwidth() (*méthode wave.Wave_read*), 1550
getscreen() (*dans le module turtle*), 1594
getservbyname() (*dans le module socket*), 1175
getservbyport() (*dans le module socket*), 1175
GetSetDescriptorType (*dans le module types*), 303
getshapes() (*dans le module turtle*), 1600
getsid() (*dans le module os*), 671
getsignal() (*dans le module signal*), 1238
getsitepackages() (*dans le module site*), 2051
getsize() (*dans le module os.path*), 474
getsizeof() (*dans le module sys*), 1940
getsockname() (*méthode socket.socket*), 1180
getsockopt() (*méthode socket.socket*), 1180
getsource() (*dans le module inspect*), 2031
getsourcefile() (*dans le module inspect*), 2031
getsourcelines() (*dans le module inspect*), 2031
getstate() (*dans le module random*), 384
getstate() (*méthode codecs.IncrementalDecoder*), 195
getstate() (*méthode codecs.IncrementalEncoder*), 195
getstate() (*méthode random.Random*), 388
getstatusoutput() (*dans le module subprocess*), 1046
getstr() (*méthode curses.window*), 933
getSubject() (*méthode logging.handlers.SMTPHandler*), 796
getswitchinterval() (*dans le module sys*), 1941

getSystemId() (*méthode xml.sax.xmlreader.InputSource*), 1401
getSystemId() (*méthode xml.sax.xmlreader.Locator*), 1401
getsyx() (*dans le module curses*), 926
gettarinfo() (*méthode tarfile.TarFile*), 602
gettempdir() (*dans le module tempfile*), 490
gettempdirb() (*dans le module tempfile*), 490
gettempprefix() (*dans le module tempfile*), 490
gettempprefixb() (*dans le module tempfile*), 490
getTestCaseNames() (*méthode unittest.TestLoader*), 1775
gettext
    module, 1553
gettext() (*dans le module gettext*), 1554
gettext() (*dans le module locale*), 1569
gettext() (*méthode gettext.GNUTranslations*), 1557
gettext() (*méthode gettext.NullTranslations*), 1555
gettimeout() (*méthode socket.socket*), 1180
gettrace() (*dans le module sys*), 1941
gettrace() (*dans le module threading*), 957
getturtle() (*dans le module turtle*), 1594
getType() (*méthode xml.sax.xmlreader.Attributes*), 1402
getuid() (*dans le module os*), 669
getunicodeinternedsize() (*dans le module sys*), 1939
geturl() (*méthode http.client.HTTPResponse*), 1463
geturl() (*méthode urllib.parse.urllib.parse.SplitResult*), 1449
geturl() (*méthode urllib.response.addinfourl*), 1442
getuser() (*dans le module getpass*), 921
getuserbase() (*dans le module site*), 2051
getusersitepackages() (*dans le module site*), 2051
getvalue() (*méthode io.BytesIO*), 738
getvalue() (*méthode io.StringIO*), 742
getValue() (*méthode xml.sax.xmlreader.Attributes*), 1402
getValueByQName() (*méthode xml.sax.xmlreader.AttributesNS*), 1402
getwch() (*dans le module msvcrt*), 2184
getwche() (*dans le module msvcrt*), 2184
getweakrefcount() (*dans le module weakref*), 293
getweakrefs() (*dans le module weakref*), 293
getwelcome() (*méthode ftplib.FTP*), 1466
getwelcome() (*méthode poplib.POP3*), 1472
getwin() (*dans le module curses*), 926
getwindowsversion() (*dans le module sys*), 1941
getwriter() (*dans le module codecs*), 190
getxattr() (*dans le module os*), 711
getyx() (*méthode curses.window*), 934
gid (*attribut tarfile.TarInfo*), 604
GIL, **2236**
glob
    module, 492, 495
glob() (*dans le module glob*), 492
glob() (*méthode pathlib.Path*), 464
Global (*classe dans ast*), 2132
global_enum() (*dans le module enum*), 330
globals()
    built-in function, 18
globs (*attribut doctest.DocTest*), 1743
gmtime() (*dans le module time*), 746
gname (*attribut tarfile.TarInfo*), 604
GNOME, 1557
GNU_FORMAT (*dans le module tarfile*), 598
gnu_getopt() (*dans le module getopt*), 2218
GNUTranslations (*classe dans gettext*), 1557
GNUTYPE_LONGLINK (*dans le module tarfile*), 598
GNUTYPE_LONGNAME (*dans le module tarfile*), 598
GNUTYPE_SPARSE (*dans le module tarfile*), 598
go() (*méthode tkinter.filedialog.FileDialog*), 1636
got (*attribut doctest.DocTestFailure*), 1750
goto() (*dans le module turtle*), 1579
grantpt() (*dans le module os*), 676
Graphical User Interface, 1619
graphlib
    module, 331
GREATER (*dans le module token*), 2144
GREATEREQUAL (*dans le module token*), 2145
Greenwich Mean Time, 744
GRND_NONBLOCK (*dans le module os*), 730
GRND_RANDOM (*dans le module os*), 730
Group (*classe dans email.headerregistry*), 1278
group() (*méthode pathlib.Path*), 469
group() (*méthode re.Match*), 148
groupby() (*dans le module itertools*), 415
groupdict() (*méthode re.Match*), 149
groupindex (*attribut re.Pattern*), 147
groups (*attribut email.headerregistry.AddressHeader*), 1275
groups (*attribut re.Pattern*), 147
groups() (*méthode re.Match*), 149
grp
    module, 2199
GS (*dans le module curses.ascii*), 951
Gt (*classe dans ast*), 2111
gt() (*dans le module operator*), 437
GtE (*classe dans ast*), 2111
guess_all_extensions() (*dans le module mimetypes*), 1336
guess_all_extensions() (*méthode mimetypes.MimeTypes*), 1338
guess_extension() (*dans le module mimetypes*), 1337
guess_extension() (*méthode mimetypes.MimeTypes*), 1338
guess_file_type() (*dans le module mimetypes*), 1336
guess_file_type() (*méthode mimetypes.MimeTypes*), 1338
guess_scheme() (*dans le module wsgiref.util*), 1416
guess_type() (*dans le module mimetypes*), 1336
guess_type() (*méthode mimetypes.MimeTypes*), 1338
GUI, 1619
gzip

module, 571
GzipFile (*classe dans gzip*), 571

H

-h
option de ligne de commande ast, 2138
option de ligne de commande calendar, 257
option de ligne de commande dis, 2160
option de ligne de commande gzip, 574
option de ligne de commande json, 1317
option de ligne de commande python--m-sqlite3-[-h]-[-v]-[filename]-[sql], 558
option de ligne de commande random, 392
option de ligne de commande timeit, 1896
option de ligne de commande tokenize, 2149
option de ligne de commande uuid, 1492
option de ligne de commande zipapp, 1925
hachable, **2236**
halfdelay() (*dans le module curses*), 926
Handle (*classe dans asyncio*), 1125
handle() (*méthode http.server.BaseHTTPRequestHandler*), 1504
handle() (*méthode logging.Handler*), 763
handle() (*méthode logging.handlers.QueueListener*), 799
handle() (*méthode logging.Logger*), 761
handle() (*méthode logging.NullHandler*), 787
handle() (*méthode socketserver.BaseRequestHandler*), 1498
handle() (*méthode wsgiref.simple_server.WSGIRequestHandler*), 1420
handle_charref() (*méthode html.parser.HTMLParser*), 1349
handle_comment() (*méthode html.parser.HTMLParser*), 1350
handle_data() (*méthode html.parser.HTMLParser*), 1349
handle_decl() (*méthode html.parser.HTMLParser*), 1350
handle_defect() (*méthode email.policy.Policy*), 1267
handle_endtag() (*méthode html.parser.HTMLParser*), 1349
handle_entityref() (*méthode html.parser.HTMLParser*), 1349
handle_error() (*méthode socketserver.BaseServer*), 1497
handle_expect_100() (*méthode http.server.BaseHTTPRequestHandler*), 1504
handle_one_request() (*méthode http.server.BaseHTTPRequestHandler*), 1504
handle_pi() (*méthode html.parser.HTMLParser*), 1350
handle_request() (*méthode socketserver.BaseServer*), 1496
handle_request() (*méthode xmlrpc.server.CGIXMLRPCRequestHandler*), 1533
handle_startendtag() (*méthode html.parser.HTMLParser*), 1349
handle_starttag() (*méthode html.parser.HTMLParser*), 1349
handle_timeout() (*méthode socketserver.BaseServer*), 1497
handleError() (*méthode logging.Handler*), 763
handleError() (*méthode logging.handlers.SocketHandler*), 792
Handler (*classe dans logging*), 762
handlers (*attribut logging.Logger*), 758
Handlers (*classe dans signal*), 1235
hardlink_to() (*méthode pathlib.Path*), 467
--hardlink-dupes
option de ligne de commande compileall, 2157
harmonic_mean() (*dans le module statistics*), 396
HAS_ALPN (*dans le module ssl*), 1199
has_children() (*méthode symtable.SymbolTable*), 2140
has_colors() (*dans le module curses*), 926
has_default() (*méthode typing.ParamSpec*), 1701
has_default() (*méthode typing.TypeVar*), 1697
has_default() (*méthode typing.TypeVarTuple*), 1699
has_dualstack_ipv6() (*dans le module socket*), 1172
HAS_ECDH (*dans le module ssl*), 1199
has_extended_color_support() (*dans le module curses*), 926
has_extn() (*méthode smtplib.SMTP*), 1485
has_header() (*méthode csv.Sniffer*), 616
has_header() (*méthode urllib.request.Request*), 1432
has_ic() (*dans le module curses*), 926
has_il() (*dans le module curses*), 926
has_ipv6 (*dans le module socket*), 1169
has_key() (*dans le module curses*), 926
has_location (*attribut importlib.machinery.ModuleSpec*), 2079
HAS_NEVER_CHECK_COMMON_NAME (*dans le module ssl*), 1199
has_nonstandard_attr() (*méthode http.cookiejar.Cookie*), 1520
HAS_NPN (*dans le module ssl*), 1199
has_option() (*méthode configparser.ConfigParser*), 635
has_option() (*méthode optparse.OptionParser*), 912
HAS_PHA (*dans le module ssl*), 1200
HAS_PSK (*dans le module ssl*), 1200
has_section() (*méthode configparser.ConfigParser*), 635

HAS_SNI (*dans le module ssl*), 1199
HAS_SSLv2 (*dans le module ssl*), 1200
HAS_SSLv3 (*dans le module ssl*), 1200
has_ticket (*attribut ssl.SSLSession*), 1222
HAS_TLSv1 (*dans le module ssl*), 1200
HAS_TLSv1_1 (*dans le module ssl*), 1200
HAS_TLSv1_2 (*dans le module ssl*), 1200
HAS_TLSv1_3 (*dans le module ssl*), 1200
hasarg (*dans le module dis*), 2180
hasattr()
    built-in function, 18
hasAttribute() (*méthode xml.dom.Element*), 1380
hasAttributeNS() (*méthode xml.dom.Element*), 1380
hasAttributes() (*méthode xml.dom.Node*), 1377
hasChildNodes() (*méthode xml.dom.Node*), 1377
hascompare (*dans le module dis*), 2181
hasconst (*dans le module dis*), 2181
hasexc (*dans le module dis*), 2181
hasFeature() (*méthode xml.dom.DOMImplementation*), 1376
hasfree (*dans le module dis*), 2181
hash
    built-in function, 48
hash()
    built-in function, 18
hash_bits (*attribut sys.hash_info*), 1943
hash_info (*dans le module sys*), 1942
hash_randomization (*attribut sys.flags*), 1936
Hashable (*classe dans collections.abc*), 279
Hashable (*classe dans typing*), 1723
hasHandlers() (*méthode logging.Logger*), 761
hash.block_size (*dans le module hashlib*), 647
hash.digest_size (*dans le module hashlib*), 647
hashlib
    module, 645
hasjabs (*dans le module dis*), 2181
hasjrel (*dans le module dis*), 2181
hasjump (*dans le module dis*), 2181
haslocal (*dans le module dis*), 2181
hasname (*dans le module dis*), 2181
HAVE_ARGUMENT (*opcode*), 2178
HAVE_CONTEXTVAR (*dans le module decimal*), 370
HAVE_DOCSTRINGS (*dans le module test.support*), 1848
HAVE_THREADS (*dans le module decimal*), 369
HCI_DATA_DIR (*dans le module socket*), 1170
HCI_FILTER (*dans le module socket*), 1170
HCI_TIME_STAMP (*dans le module socket*), 1170
Header (*classe dans email.header*), 1299
header_encode() (*méthode email.charset.Charset*), 1302
header_encode_lines() (*méthode email.charset.Charset*), 1302
header_encoding (*attribut email.charset.Charset*), 1301
header_factory (*attribut email.policy.EmailPolicy*), 1269
header_fetch_parse() (*méthode email.policy.Compat32*), 1271
header_fetch_parse() (*méthode email.policy.EmailPolicy*), 1270
header_fetch_parse() (*méthode email.policy.Policy*), 1268
header_items() (*méthode urllib.request.Request*), 1433
header_max_count() (*méthode email.policy.EmailPolicy*), 1269
header_max_count() (*méthode email.policy.Policy*), 1268
header_offset (*attribut zipfile.ZipInfo*), 594
header_source_parse() (*méthode email.policy.Compat32*), 1271
header_source_parse() (*méthode email.policy.EmailPolicy*), 1270
header_source_parse() (*méthode email.policy.Policy*), 1268
header_store_parse() (*méthode email.policy.Compat32*), 1271
header_store_parse() (*méthode email.policy.EmailPolicy*), 1270
header_store_parse() (*méthode email.policy.Policy*), 1268
HeaderDefect, 1272
HeaderError, 597
HeaderParseError, 1272
HeaderParser (*classe dans email.parser*), 1261
HeaderRegistry (*classe dans email.headerregistry*), 1276
headers
    MIME, 1336
headers (*attribut http.client.HTTPResponse*), 1463
headers (*attribut http.server.BaseHTTPRequestHandler*), 1503
headers (*attribut urllib.error.HTTPError*), 1452
headers (*attribut urllib.response.addinfourl*), 1442
headers (*attribut xmlrpc.client.ProtocolError*), 1526
Headers (*classe dans wsgiref.headers*), 1418
HeaderWriteError, 1272
heading() (*dans le module turtle*), 1584
heading() (*méthode tkinter.ttk.Treeview*), 1652
heapify() (*dans le module heapq*), 282
heapmin() (*dans le module msvcrt*), 2184
heappop() (*dans le module heapq*), 282
heappush() (*dans le module heapq*), 282
heappushpop() (*dans le module heapq*), 282
heapq
    module, 282
heapreplace() (*dans le module heapq*), 282
helo() (*méthode smtplib.SMTP*), 1484
help
    online, 1724
--help
    option de ligne de commande ast, 2138
    option de ligne de commande calendar, 257
    option de ligne de commande dis, 2160
    option de ligne de commande gzip, 574

option de ligne de commande json, 1317
option de ligne de commande python--m-sqlite3-[-h]-[-v]-[filename]-[sql], 558
option de ligne de commande random, 392
option de ligne de commande timeit, 1896
option de ligne de commande tokenize, 2149
option de ligne de commande trace, 1899
option de ligne de commande uuid, 1492
option de ligne de commande zipapp, 1925
help (*attribut optparse.Option*), 908
help (*pdb command*), 1880
help()
built-in function, 18
herror, 1165
hex (*attribut uuid.UUID*), 1490
hex()
built-in function, 18
hex() (*méthode bytearray*), 66
hex() (*méthode bytes*), 65
hex() (*méthode float*), 43
hex() (*méthode memoryview*), 83
hexadecimal
literals, 39
hexdigest() (*méthode hashlib.hash*), 648
hexdigest() (*méthode hashlib.shake*), 648
hexdigest() (*méthode hmac.HMAC*), 657
hexdigits (*dans le module string*), 121
hexlify() (*dans le module binascii*), 1344
hexversion (*dans le module sys*), 1943
hidden() (*méthode curses.panel.Panel*), 954
hide() (*méthode curses.panel.Panel*), 954
hide() (*méthode tkinter.ttk.Notebook*), 1647
hide_cookie2 (*attribut http.cookiejar.CookiePolicy*), 1517
hideturtle() (*dans le module turtle*), 1589
HierarchyRequestErr, 1382
HIGH_PRIORITY_CLASS (*dans le module subprocess*), 1041
HIGHEST_PROTOCOL (*dans le module pickle*), 511
hits (*attribut bdb.Breakpoint*), 1870
HKEY_CLASSES_ROOT (*dans le module winreg*), 2191
HKEY_CURRENT_CONFIG (*dans le module winreg*), 2191
HKEY_CURRENT_USER (*dans le module winreg*), 2191
HKEY_DYN_DATA (*dans le module winreg*), 2192
HKEY_LOCAL_MACHINE (*dans le module winreg*), 2191
HKEY_PERFORMANCE_DATA (*dans le module winreg*), 2191
HKEY_USERS (*dans le module winreg*), 2191
hline() (*méthode curses.window*), 934
hls_to_rgb() (*dans le module colorsys*), 1552
hmac
module, 657
HOME, 474, 1621
home() (*dans le module turtle*), 1581
home() (*méthode de la classe pathlib.Path*), 459
HOMEDRIVE, 474
HOMEPATH, 474
hook_compressed() (*dans le module fileinput*), 923
hook_encoded() (*dans le module fileinput*), 923
host (*attribut urllib.request.Request*), 1431
hostmask (*attribut ipaddress.IPv4Network*), 1541
hostmask (*attribut ipaddress.IPv6Network*), 1544
hostname_checks_common_name (*attribut ssl.SSLContext*), 1213
hosts (*attribut netrc.netrc*), 642
hosts() (*méthode ipaddress.IPv4Network*), 1541
hosts() (*méthode ipaddress.IPv6Network*), 1544
hour (*attribut datetime.datetime*), 222
hour (*attribut datetime.time*), 230
hresult (*attribut ctypes.COMError*), 848
HRESULT (*classe dans ctypes*), 845
hStdError (*attribut subprocess.STARTUPINFO*), 1040
hStdInput (*attribut subprocess.STARTUPINFO*), 1040
hStdOutput (*attribut subprocess.STARTUPINFO*), 1040
hsv_to_rgb() (*dans le module colorsys*), 1552
HT (*dans le module curses.ascii*), 950
ht() (*dans le module turtle*), 1589
HTML, 1347, 1442
html
module, 1347
html5 (*dans le module html.entities*), 1352
HTMLCalendar (*classe dans calendar*), 252
HtmlDiff (*classe dans difflib*), 156
html.entities
module, 1352
html.parser
module, 1347
HTMLParser (*classe dans html.parser*), 1348
htonl() (*dans le module socket*), 1175
htons() (*dans le module socket*), 1175
HTTP
http (*standard module*), 1454
http.client (*standard module*), 1457
protocol, 1442, 1454, 1457, 1502
http
module, 1454
HTTP (*dans le module email.policy*), 1270
http_error_301() (*méthode urllib.request.HTTPRedirectHandler*), 1435
http_error_302() (*méthode urllib.request.HTTPRedirectHandler*), 1435
http_error_303() (*méthode urllib.request.HTTPRedirectHandler*), 1435
http_error_307() (*méthode urllib.request.HTTPRedirectHandler*), 1435
http_error_308() (*méthode urllib.request.HTTPRedirectHandler*), 1435
http_error_401() (*méthode urllib.request.HTTPBasicAuthHandler*), 1437

http_error_401() (*méthode urllib.request.HTTPDigestAuthHandler*), 1437
http_error_407() (*méthode urllib.request.ProxyBasicAuthHandler*), 1437
http_error_407() (*méthode urllib.request.ProxyDigestAuthHandler*), 1437
http_error_auth_reqed() (*méthode urllib.request.AbstractBasicAuthHandler*), 1436
http_error_auth_reqed() (*méthode urllib.request.AbstractDigestAuthHandler*), 1437
http_error_default() (*méthode urllib.request.BaseHandler*), 1434
http_open() (*méthode urllib.request.HTTPHandler*), 1437
HTTP_PORT (*dans le module http.client*), 1459
http_response() (*méthode urllib.request.HTTPErrorProcessor*), 1438
http_version (*attribut wsgiref.handlers.BaseHandler*), 1424
HTTPBasicAuthHandler (*classe dans urllib.request*), 1430
http.client
    module, 1457
HTTPConnection (*classe dans http.client*), 1457
http.cookiejar
    module, 1512
HTTPCookieProcessor (*classe dans urllib.request*), 1429
http.cookies
    module, 1509
httpd, 1502
HTTPDefaultErrorHandler (*classe dans urllib.request*), 1429
HTTPDigestAuthHandler (*classe dans urllib.request*), 1431
HTTPError, 1452
HTTPErrorProcessor (*classe dans urllib.request*), 1431
HTTPException, 1458
HTTPHandler (*classe dans logging.handlers*), 797
HTTPHandler (*classe dans urllib.request*), 1431
HTTPMessage (*classe dans http.client*), 1465
HTTPMethod (*classe dans http*), 1456
httponly (*attribut http.cookies.Morsel*), 1510
HTTPPasswordMgr (*classe dans urllib.request*), 1430
HTTPPasswordMgrWithDefaultRealm (*classe dans urllib.request*), 1430
HTTPPasswordMgrWithPriorAuth (*classe dans urllib.request*), 1430
HTTPRedirectHandler (*classe dans urllib.request*), 1429
HTTPResponse (*classe dans http.client*), 1458
https_open() (*méthode urllib.request.HTTPSHandler*), 1437
HTTPS_PORT (*dans le module http.client*), 1459
https_response() (*méthode urllib.request.HTTPErrorProcessor*), 1438
HTTPSConnection (*classe dans http.client*), 1458
http.server
    module, 1502
    security, 1508
HTTPServer (*classe dans http.server*), 1503
HTTPSHandler (*classe dans urllib.request*), 1431
HTTPStatus (*classe dans http*), 1454
HV_GUID_BROADCAST (*dans le module socket*), 1170
HV_GUID_CHILDREN (*dans le module socket*), 1170
HV_GUID_LOOPBACK (*dans le module socket*), 1170
HV_GUID_PARENT (*dans le module socket*), 1170
HV_GUID_WILDCARD (*dans le module socket*), 1170
HV_GUID_ZERO (*dans le module socket*), 1170
HV_PROTOCOL_RAW (*dans le module socket*), 1170
HVSOCKET_ADDRESS_FLAG_PASSTHRU (*dans le module socket*), 1170
HVSOCKET_CONNECT_TIMEOUT (*dans le module socket*), 1170
HVSOCKET_CONNECT_TIMEOUT_MAX (*dans le module socket*), 1170
HVSOCKET_CONNECTED_SUSPEND (*dans le module socket*), 1170
hypot() (*dans le module math*), 344

I

-i
    option de ligne de commande ast, 2139
    option de ligne de commande compileall, 2156
    option de ligne de commande random, 392
I (*dans le module re*), 140
I/O control
    buffering, 25, 1181
    POSIX, 2200
    tty, 2200
    UNIX, 2204
iadd() (*dans le module operator*), 443
iand() (*dans le module operator*), 443
iconcat() (*dans le module operator*), 443
id (*attribut ssl.SSLSession*), 1222
id()
    built-in function, 19
id() (*méthode unittest.TestCase*), 1770
idcok() (*méthode curses.window*), 934
ident (*attribut select.kevent*), 1229
ident (*attribut threading.Thread*), 960
identchars (*attribut cmd.Cmd*), 1610
identify() (*méthode tkinter.ttk.Notebook*), 1647
identify() (*méthode tkinter.ttk.Treeview*), 1652
identify() (*méthode tkinter.ttk.Widget*), 1644
identify_column() (*méthode tkinter.ttk.Treeview*), 1652
identify_element() (*méthode tkinter.ttk.Treeview*), 1653
identify_region() (*méthode tkinter.ttk.Treeview*), 1652
identify_row() (*méthode tkinter.ttk.Treeview*), 1652

IDLE, **2236**
idle() (*méthode imaplib.IMAP4*), 1477
IDLE*STARTUP, 1666
IDLE_PRIORITY_CLASS (*dans le module subprocess*), 1041
idlelib
    module, 1670
idlok() (*méthode curses.window*), 934
if
    statement, 37
If (*classe dans ast*), 2119
if_indextoname() (*dans le module socket*), 1178
if_nameindex() (*dans le module socket*), 1177
if_nametoindex() (*dans le module socket*), 1177
IfExp (*classe dans ast*), 2112
ifloordiv() (*dans le module operator*), 443
iglob() (*dans le module glob*), 493
ignorableWhitespace() (*méthode xml.sax.handler.ContentHandler*), 1396
ignore
    error handler's name, 192
ignore (*attribut bdb.Breakpoint*), 1870
IGNORE (*dans le module tkinter.messagebox*), 1638
ignore (*pdb command*), 1881
ignore_environment (*attribut sys.flags*), 1936
ignore_errors() (*dans le module codecs*), 193
IGNORE_EXCEPTION_DETAIL (*dans le module doctest*), 1736
ignore_patterns() (*dans le module shutil*), 499
ignore_warnings() (*dans le module test.support.warnings_helper*), 1862
IGNORECASE (*dans le module re*), 140
--ignore-dir
    option de ligne de commande trace, 1900
--ignore-module
    option de ligne de commande trace, 1900
IISCGIHandler (*classe dans wsgiref.handlers*), 1421
IllegalMonthError, 256
IllegalWeekdayError, 256
ilshift() (*dans le module operator*), 443
imag (*attribut numbers.Complex*), 335
imag (*attribut sys.hash_info*), 1943
imap() (*méthode multiprocessing.pool.Pool*), 999
IMAP4
    protocol, 1474
IMAP4 (*classe dans imaplib*), 1474
IMAP4_SSL
    protocol, 1474
IMAP4_SSL (*classe dans imaplib*), 1475
IMAP4_stream
    protocol, 1474
IMAP4_stream (*classe dans imaplib*), 1475
IMAP4.abort, 1474
IMAP4.error, 1474
IMAP4.readonly, 1474
imap_unordered() (*méthode multiprocessing.pool.Pool*), 999
imaplib
    module, 1474
imatmul() (*dans le module operator*), 443
imghdr
    module, 2223
immedok() (*méthode curses.window*), 934
immortal, **2236**
immuable, **2236**
immutable
    sequence types, 48
imod() (*dans le module operator*), 443
imp
    module, 2223
impl_detail() (*dans le module test.support*), 1853
implementation (*dans le module sys*), 1943
import
    statement, 33, 2049
Import (*classe dans ast*), 2118
import_fresh_module() (*dans le module test.support.import_helper*), 1861
IMPORT_FROM (*opcode*), 2174
import_module() (*dans le module importlib*), 2067
import_module() (*dans le module test.support.import_helper*), 1861
IMPORT_NAME (*opcode*), 2173
importateur, **2236**
importation, **2236**
ImportError, 109
ImportFrom (*classe dans ast*), 2118
importlib
    module, 2066
importlib.abc
    module, 2068
importlib.machinery
    module, 2074
importlib.metadata
    module, 2090
importlib.resources
    module, 2085
importlib.resources.abc
    module, 2088
importlib.util
    module, 2079
ImportWarning, 117
ImproperConnectionState, 1459
imul() (*dans le module operator*), 443
in
    operator, 38, 47
In (*classe dans ast*), 2111
in_dll() (*méthode ctypes._CData*), 842
in_table_a1() (*dans le module stringprep*), 174
in_table_b1() (*dans le module stringprep*), 174
in_table_c3() (*dans le module stringprep*), 175
in_table_c4() (*dans le module stringprep*), 175
in_table_c5() (*dans le module stringprep*), 175
in_table_c6() (*dans le module stringprep*), 175

in_table_c7() (*dans le module stringprep*), 175
in_table_c8() (*dans le module stringprep*), 175
in_table_c9() (*dans le module stringprep*), 175
in_table_c11() (*dans le module stringprep*), 174
in_table_c11_c12() (*dans le module stringprep*), 175
in_table_c12() (*dans le module stringprep*), 175
in_table_c21() (*dans le module stringprep*), 175
in_table_c21_c22() (*dans le module stringprep*), 175
in_table_c22() (*dans le module stringprep*), 175
in_table_d1() (*dans le module stringprep*), 175
in_table_d2() (*dans le module stringprep*), 175
in_transaction (*attribut sqlite3.Connection*), 551
inch() (*méthode curses.window*), 934
include() (*dans le module xml.etree.ElementInclude*), 1366
--include-attributes
    option de ligne de commande ast, 2139
inclusive (*attribut tracemalloc.DomainFilter*), 1907
inclusive (*attribut tracemalloc.Filter*), 1907
Incomplete, 1344
IncompleteRead, 1459
IncompleteReadError, 1105
increment_lineno() (*dans le module ast*), 2136
incrementaldecoder (*attribut codecs.CodecInfo*), 189
IncrementalDecoder (*classe dans codecs*), 195
incrementalencoder (*attribut codecs.CodecInfo*), 189
IncrementalEncoder (*classe dans codecs*), 195
IncrementalNewlineDecoder (*classe dans io*), 743
IncrementalParser (*classe dans xml.sax.xmlreader*), 1399
--indent
    option de ligne de commande ast, 2139
    option de ligne de commande json, 1317
indent (*attribut doctest.Example*), 1744
indent (*attribut reprlib.Repr*), 314
INDENT (*dans le module token*), 2144
indent() (*dans le module textwrap*), 169
indent() (*dans le module xml.etree.ElementTree*), 1363
IndentationError, 113
--indentlevel
    option de ligne de commande pickletools, 2182
index (*attribut inspect.FrameInfo*), 2039
index (*attribut inspect.Traceback*), 2039
index() (*dans le module operator*), 438
index() (*méthode array.array*), 291
index() (*méthode bytearray*), 70
index() (*méthode bytes*), 70
index() (*méthode collections.deque*), 265
index() (*méthode memoryview*), 86
index() (*méthode multiprocessing.shared_memory.ShareableList*), 1018
index() (*méthode str*), 56
index() (*méthode tkinter.ttk.Notebook*), 1647
index() (*méthode tkinter.ttk.Treeview*), 1653
index() (*sequence method*), 47
IndexError, 110
indexOf() (*dans le module operator*), 440
IndexSizeErr, 1382
indication de type, **2244**
INDIRECT (*attribut inspect.BufferFlags*), 2044
inet_aton() (*dans le module socket*), 1176
inet_ntoa() (*dans le module socket*), 1176
inet_ntop() (*dans le module socket*), 1176
inet_pton() (*dans le module socket*), 1176
Inexact (*classe dans decimal*), 371
inf (*attribut sys.hash_info*), 1942
inf (*dans le module cmath*), 350
inf (*dans le module math*), 346
infile
    option de ligne de commande json, 1316
infile (*attribut shlex.shlex*), 1616
Infinity, 16
infj (*dans le module cmath*), 350
--info
    option de ligne de commande zipapp, 1925
info (*attribut pathlib.Path*), 462
INFO (*dans le module logging*), 762
INFO (*dans le module tkinter.messagebox*), 1639
info() (*dans le module logging*), 770
info() (*méthode dis.Bytecode*), 2161
info() (*méthode gettext.NullTranslations*), 1556
info() (*méthode http.client.HTTPResponse*), 1463
info() (*méthode logging.Logger*), 760
info() (*méthode urllib.response.addinfourl*), 1442
infolist() (*méthode zipfile.ZipFile*), 587
.ini
    file, 620
ini file, 620
init() (*dans le module mimetypes*), 1337
init_color() (*dans le module curses*), 926
init_pair() (*dans le module curses*), 926
inited (*dans le module mimetypes*), 1337
initgroups() (*dans le module os*), 669
initial_indent (*attribut textwrap.TextWrapper*), 171
initscr() (*dans le module curses*), 926
inode() (*méthode os.DirEntry*), 696
input()
    built-in function, 19
input() (*dans le module fileinput*), 921
input_charset (*attribut email.charset.Charset*), 1301
input_codec (*attribut email.charset.Charset*), 1302
InputSource (*classe dans xml.sax.xmlreader*), 1399
InputStream (*classe dans wsgiref.types*), 1424
insch() (*méthode curses.window*), 934
insdelln() (*méthode curses.window*), 934
insert() (*méthode array.array*), 291
insert() (*méthode collections.deque*), 265
insert() (*méthode tkinter.ttk.Notebook*), 1647
insert() (*méthode tkinter.ttk.Treeview*), 1653
insert() (*méthode xml.etree.ElementTree.Element*), 1367
insert() (*sequence method*), 49
insert_text() (*dans le module readline*), 176
insertBefore() (*méthode xml.dom.Node*), 1378

insertln() (*méthode curses.window*), 934
insnstr() (*méthode curses.window*), 934
insort() (*dans le module bisect*), 286
insort_left() (*dans le module bisect*), 286
insort_right() (*dans le module bisect*), 286
inspect
    module, 2025
inspect (*attribut sys.flags*), 1936
InspectLoader (*classe dans importlib.abc*), 2070
insstr() (*méthode curses.window*), 934
install() (*dans le module gettext*), 1555
install() (*méthode gettext.NullTranslations*), 1556
install_opener() (*dans le module urllib.request*), 1427
install_scripts() (*méthode venv.EnvBuilder*), 1920
installHandler() (*dans le module unittest*), 1783
instate() (*méthode tkinter.ttk.Widget*), 1644
instr() (*méthode curses.window*), 934
instream (*attribut shlex.shlex*), 1616
instruction, **2243**
Instruction (*classe dans dis*), 2164
INSTRUCTION (*monitoring event*), 1956
Instruction.arg (*dans le module dis*), 2165
Instruction.argrepr (*dans le module dis*), 2165
Instruction.argval (*dans le module dis*), 2165
Instruction.baseopcode (*dans le module dis*), 2164
Instruction.baseopname (*dans le module dis*), 2165
Instruction.cache_offset (*dans le module dis*), 2165
Instruction.end_offset (*dans le module dis*), 2165
Instruction.is_jump_target (*dans le module dis*), 2165
Instruction.jump_target (*dans le module dis*), 2165
Instruction.line_number (*dans le module dis*), 2165
Instruction.offset (*dans le module dis*), 2165
Instruction.oparg (*dans le module dis*), 2165
Instruction.opcode (*dans le module dis*), 2164
Instruction.opname (*dans le module dis*), 2164
Instruction.positions (*dans le module dis*), 2165
Instruction.start_offset (*dans le module dis*), 2165
Instruction.starts_line (*dans le module dis*), 2165
int
    built-in function, 39
int (*attribut uuid.UUID*), 1490
int (*classe de base*), 19
Int2AP() (*dans le module imaplib*), 1475
int_info (*dans le module sys*), 1944
int_max_str_digits (*attribut sys.flags*), 1936
integer
    literals, 39
    object, 38
    types, operations on, 40
--integer
    option de ligne de commande random, 392
Integral (*classe dans numbers*), 336
Integrated Development Environment, 1659
IntegrityError, 556
IntEnum (*classe dans enum*), 322
interact (*pdb command*), 1883
interact() (*dans le module code*), 2053
interact() (*méthode code.InteractiveConsole*), 2054
interactif, **2236**
interactive (*attribut sys.flags*), 1936
Interactive (*classe dans ast*), 2106
InteractiveConsole (*classe dans code*), 2053
InteractiveInterpreter (*classe dans code*), 2053
InterfaceError, 556
intern() (*dans le module sys*), 1944
internal_attr (*attribut zipfile.ZipInfo*), 594
Internaldate2tuple() (*dans le module imaplib*), 1475
InternalError, 556
internalSubset (*attribut xml.dom.DocumentType*), 1379
Internet, 1413
INTERNET_TIMEOUT (*dans le module test.support*), 1847
interpolation
    bytearray (%), 79
    bytes (%), 79
interpolation, string (%), 63
InterpolationDepthError, 638
InterpolationError, 638
InterpolationMissingOptionError, 639
InterpolationSyntaxError, 639
interprété, **2237**
interpreter prompts, 1948
interpreter_requires_environment() (*dans le module test.support.script_helper*), 1857
InterpreterPoolExecutor (*classe dans concurrent.futures*), 1023
interrupt() (*méthode sqlite3.Connection*), 546
interrupt_main() (*dans le module _thread*), 1057
InterruptedError, 115
intersection() (*méthode frozenset*), 89
intersection_update() (*méthode frozenset*), 90
IntFlag (*classe dans enum*), 325
intro (*attribut cmd.Cmd*), 1610
InuseAttributeErr, 1382
inv() (*dans le module operator*), 438
inv_cdf() (*méthode statistics.NormalDist*), 405
InvalidAccessErr, 1382
invalidate_caches() (*dans le module importlib*), 2067
invalidate_caches() (*dans le module platform*), 804
invalidate_caches() (*méthode de la classe importlib.machinery.PathFinder*), 2075
invalidate_caches() (*méthode importlib.abc.MetaPathFinder*), 2068

invalidate_caches() (*méthode importlib.abc.PathEntryFinder*), 2069
invalidate_caches() (*méthode importlib.machinery.FileFinder*), 2076
invalidate_caches() (*méthode zipimport.zipimporter*), 2059
--invalidation-mode
    option de ligne de commande compileall, 2157
InvalidCharacterErr, 1382
InvalidModificationErr, 1382
InvalidOperation (*classe dans decimal*), 371
InvalidStateErr, 1382
InvalidStateError, 1027, 1105
InvalidTZPathWarning, 250
InvalidURL, 1459
Invert (*classe dans ast*), 2110
invert() (*dans le module operator*), 438
io
    module, 730
IO (*classe dans typing*), 1710
IO_REPARSE_TAG_APPEXECLINK (*dans le module stat*), 484
IO_REPARSE_TAG_MOUNT_POINT (*dans le module stat*), 484
IO_REPARSE_TAG_SYMLINK (*dans le module stat*), 484
IOBase (*classe dans io*), 734
ioctl() (*dans le module fcntl*), 2205
ioctl() (*méthode socket.socket*), 1180
IOCTL_VM_SOCKETS_GET_LOCAL_CID (*dans le module socket*), 1169
IOError, 114
ior() (*dans le module operator*), 443
ios_ver() (*dans le module platform*), 803
io.StringIO
    object, 53
ip (*attribut ipaddress.IPv4Interface*), 1545
ip (*attribut ipaddress.IPv6Interface*), 1546
ip_address() (*dans le module ipaddress*), 1534
ip_interface() (*dans le module ipaddress*), 1535
ip_network() (*dans le module ipaddress*), 1534
ipaddress
    module, 1534
ipow() (*dans le module operator*), 443
ipv4_mapped (*attribut ipaddress.IPv6Address*), 1538
IPv4Address (*classe dans ipaddress*), 1535
IPv4Interface (*classe dans ipaddress*), 1545
IPv4Network (*classe dans ipaddress*), 1540
IPV6_ENABLED (*dans le module test.support.socket_helper*), 1856
ipv6_mapped (*attribut ipaddress.IPv4Address*), 1537
IPv6Address (*classe dans ipaddress*), 1537
IPv6Interface (*classe dans ipaddress*), 1546
IPv6Network (*classe dans ipaddress*), 1543
irshift() (*dans le module operator*), 444
is
    operator, 38
Is (*classe dans ast*), 2111
is not
    operator, 38
is_() (*dans le module operator*), 438
is_absolute() (*méthode pathlib.PurePath*), 453
is_active() (*méthode graphlib.TopologicalSorter*), 332
is_alive() (*méthode multiprocessing.Process*), 977
is_alive() (*méthode threading.Thread*), 961
is_android (*dans le module test.support*), 1847
is_annotated() (*méthode symtable.Symbol*), 2142
is_assigned() (*méthode symtable.Symbol*), 2142
is_async (*attribut pyclbr.Function*), 2153
is_attachment() (*méthode email.message.EmailMessage*), 1255
is_authenticated() (*méthode urllib.request.HTTPPasswordMgrWithPriorAuth*), 1436
is_block_device() (*méthode pathlib.Path*), 461
is_blocked() (*méthode http.cookiejar.DefaultCookiePolicy*), 1518
is_canonical() (*méthode decimal.Context*), 367
is_canonical() (*méthode decimal.Decimal*), 359
is_char_device() (*méthode pathlib.Path*), 462
IS_CHARACTER_JUNK() (*dans le module difflib*), 160
is_check_supported() (*dans le module lzma*), 582
is_closed() (*méthode asyncio.loop*), 1109
is_closing() (*méthode asyncio.BaseTransport*), 1136
is_closing() (*méthode asyncio.StreamWriter*), 1089
is_comp_cell() (*méthode symtable.Symbol*), 2142
is_comp_iter() (*méthode symtable.Symbol*), 2142
is_dataclass() (*dans le module dataclasses*), 1984
is_declared_global() (*méthode symtable.Symbol*), 2142
is_dir() (*méthode importlib.abc.Traversable*), 2073
is_dir() (*méthode importlib.resources.abc.Traversable*), 2089
is_dir() (*méthode os.DirEntry*), 697
is_dir() (*méthode pathlib.Path*), 461
is_dir() (*méthode pathlib.types.PathInfo*), 472
is_dir() (*méthode zipfile.Path*), 591
is_dir() (*méthode zipfile.ZipInfo*), 593
is_enabled() (*dans le module faulthandler*), 1875
is_expired() (*méthode http.cookiejar.Cookie*), 1520
is_fifo() (*méthode pathlib.Path*), 461
is_file() (*méthode importlib.abc.Traversable*), 2073
is_file() (*méthode importlib.resources.abc.Traversable*), 2089
is_file() (*méthode os.DirEntry*), 697
is_file() (*méthode pathlib.Path*), 461
is_file() (*méthode pathlib.types.PathInfo*), 472
is_file() (*méthode zipfile.Path*), 591
is_finalized() (*dans le module gc*), 2024
is_finalizing() (*dans le module sys*), 1944
is_finite() (*méthode decimal.Context*), 367
is_finite() (*méthode decimal.Decimal*), 359
is_free() (*méthode symtable.Symbol*), 2142
is_free_class() (*méthode symtable.Symbol*), 2142
is_global (*attribut ipaddress.IPv4Address*), 1536

is_global (*attribut ipaddress.IPv6Address*), 1538
is_global() (*méthode symtable.Symbol*), 2142
is_hop_by_hop() (*dans le module wsgiref.util*), 1417
is_imported() (*méthode symtable.Symbol*), 2142
is_infinite() (*méthode decimal.Context*), 367
is_infinite() (*méthode decimal.Decimal*), 359
is_integer() (*méthode float*), 43
is_integer() (*méthode fractions.Fraction*), 381
is_integer() (*méthode int*), 43
is_junction() (*méthode os.DirEntry*), 697
is_junction() (*méthode pathlib.Path*), 461
is_jython (*dans le module test.support*), 1847
IS_LINE_JUNK() (*dans le module difflib*), 160
is_linetouched() (*méthode curses.window*), 934
is_link_local (*attribut ipaddress.IPv4Address*), 1537
is_link_local (*attribut ipaddress.IPv4Network*), 1541
is_link_local (*attribut ipaddress.IPv6Address*), 1538
is_link_local (*attribut ipaddress.IPv6Network*), 1543
is_local() (*méthode symtable.Symbol*), 2142
is_loopback (*attribut ipaddress.IPv4Address*), 1537
is_loopback (*attribut ipaddress.IPv4Network*), 1541
is_loopback (*attribut ipaddress.IPv6Address*), 1538
is_loopback (*attribut ipaddress.IPv6Network*), 1543
is_mount() (*méthode pathlib.Path*), 461
is_multicast (*attribut ipaddress.IPv4Address*), 1536
is_multicast (*attribut ipaddress.IPv4Network*), 1540
is_multicast (*attribut ipaddress.IPv6Address*), 1538
is_multicast (*attribut ipaddress.IPv6Network*), 1543
is_multipart() (*méthode email.message.EmailMessage*), 1251
is_multipart() (*méthode email.message.Message*), 1289
is_namespace() (*méthode symtable.Symbol*), 2143
is_nan() (*méthode decimal.Context*), 367
is_nan() (*méthode decimal.Decimal*), 359
is_nested() (*méthode symtable.SymbolTable*), 2140
is_none() (*dans le module operator*), 438
is_nonlocal() (*méthode symtable.Symbol*), 2142
is_normal() (*méthode decimal.Context*), 367
is_normal() (*méthode decimal.Decimal*), 359
is_normalized() (*dans le module unicodedata*), 173
is_not() (*dans le module operator*), 438
is_not_allowed() (*méthode http.cookiejar.DefaultCookiePolicy*), 1518
is_not_none() (*dans le module operator*), 438
IS_OP (*opcode*), 2173
is_optimized() (*méthode symtable.SymbolTable*), 2140
is_package() (*méthode importlib.abc.InspectLoader*), 2070
is_package() (*méthode importlib.abc.SourceLoader*), 2072
is_package() (*méthode importlib.machinery.ExtensionFileLoader*), 2077
is_package() (*méthode importlib.machinery.SourceFileLoader*), 2076
is_package() (*méthode importlib.machinery.SourcelessFileLoader*), 2077
is_package() (*méthode zipimport.zipimporter*), 2058
is_parameter() (*méthode symtable.Symbol*), 2142
is_private (*attribut ipaddress.IPv4Address*), 1536
is_private (*attribut ipaddress.IPv4Network*), 1540
is_private (*attribut ipaddress.IPv6Address*), 1538
is_private (*attribut ipaddress.IPv6Network*), 1543
is_protocol() (*dans le module typing*), 1717
is_python_build() (*dans le module sysconfig*), 1964
is_qnan() (*méthode decimal.Context*), 367
is_qnan() (*méthode decimal.Decimal*), 359
is_reading() (*méthode asyncio.ReadTransport*), 1136
is_referenced() (*méthode symtable.Symbol*), 2142
is_relative_to() (*méthode pathlib.PurePath*), 453
is_reserved (*attribut ipaddress.IPv4Address*), 1537
is_reserved (*attribut ipaddress.IPv4Network*), 1541
is_reserved (*attribut ipaddress.IPv6Address*), 1538
is_reserved (*attribut ipaddress.IPv6Network*), 1543
is_reserved() (*méthode pathlib.PurePath*), 454
is_resource() (*dans le module importlib.resources*), 2087
is_resource() (*méthode importlib.abc.ResourceReader*), 2073
is_resource() (*méthode importlib.resources.abc.ResourceReader*), 2088
is_resource_enabled() (*dans le module test.support*), 1849
is_running() (*méthode asyncio.loop*), 1109
is_safe (*attribut uuid.UUID*), 1490
is_serving() (*méthode asyncio.Server*), 1127
is_set() (*méthode asyncio.Event*), 1094
is_set() (*méthode threading.Event*), 966
is_signed() (*méthode decimal.Context*), 367
is_signed() (*méthode decimal.Decimal*), 359
is_site_local (*attribut ipaddress.IPv6Address*), 1538
is_site_local (*attribut ipaddress.IPv6Network*), 1544
is_skipped_line() (*méthode bdb.Bdb*), 1871
is_snan() (*méthode decimal.Context*), 367
is_snan() (*méthode decimal.Decimal*), 359
is_socket() (*méthode pathlib.Path*), 461
is_stack_trampoline_active() (*dans le module sys*), 1951
is_subnormal() (*méthode decimal.Context*), 367
is_subnormal() (*méthode decimal.Decimal*), 359
is_symlink() (*méthode os.DirEntry*), 697
is_symlink() (*méthode pathlib.Path*), 461
is_symlink() (*méthode pathlib.types.PathInfo*), 472
is_symlink() (*méthode zipfile.Path*), 591
is_tarfile() (*dans le module tarfile*), 597
is_term_resized() (*dans le module curses*), 927
is_tracing() (*dans le module tracemalloc*), 1906
is_tracked() (*dans le module gc*), 2023
is_type_parameter() (*méthode symtable.Symbol*), 2142
is_typeddict() (*dans le module typing*), 1717

is_unspecified (*attribut ipaddress.IPv4Address*), 1537
is_unspecified (*attribut ipaddress.IPv4Network*), 1540
is_unspecified (*attribut ipaddress.IPv6Address*), 1538
is_unspecified (*attribut ipaddress.IPv6Network*), 1543
is_valid() (*méthode string.Template*), 130
is_wintouched() (*méthode curses.window*), 935
is_zero() (*méthode decimal.Context*), 367
is_zero() (*méthode decimal.Decimal*), 359
is_zipfile() (*dans le module zipfile*), 585
isabs() (*dans le module os.path*), 474
isabstract() (*dans le module inspect*), 2030
IsADirectoryError, 115
isalnum() (*dans le module curses.ascii*), 952
isalnum() (*méthode bytearray*), 74
isalnum() (*méthode bytes*), 74
isalnum() (*méthode str*), 56
isalpha() (*dans le module curses.ascii*), 952
isalpha() (*méthode bytearray*), 75
isalpha() (*méthode bytes*), 75
isalpha() (*méthode str*), 56
isascii() (*dans le module curses.ascii*), 952
isascii() (*méthode bytearray*), 75
isascii() (*méthode bytes*), 75
isascii() (*méthode str*), 56
isasyncgen() (*dans le module inspect*), 2029
isasyncgenfunction() (*dans le module inspect*), 2029
isatty() (*dans le module os*), 676
isatty() (*méthode io.IOBase*), 734
isawaitable() (*dans le module inspect*), 2029
isblank() (*dans le module curses.ascii*), 952
isblk() (*méthode tarfile.TarInfo*), 605
isbuiltin() (*dans le module inspect*), 2030
ischr() (*méthode tarfile.TarInfo*), 605
isclass() (*dans le module inspect*), 2028
isclose() (*dans le module cmath*), 349
isclose() (*dans le module math*), 342
iscntrl() (*dans le module curses.ascii*), 952
iscode() (*dans le module inspect*), 2029
iscoroutine() (*dans le module asyncio*), 1080
iscoroutine() (*dans le module inspect*), 2029
iscoroutinefunction() (*dans le module inspect*), 2028
isctrl() (*dans le module curses.ascii*), 952
isDaemon() (*méthode threading.Thread*), 961
isdatadescriptor() (*dans le module inspect*), 2030
isdecimal() (*méthode str*), 56
isdev() (*méthode tarfile.TarInfo*), 605
isdevdrive() (*dans le module os.path*), 475
isdigit() (*dans le module curses.ascii*), 952
isdigit() (*méthode bytearray*), 75
isdigit() (*méthode bytes*), 75
isdigit() (*méthode str*), 57
isdir() (*dans le module os.path*), 475
isdir() (*méthode tarfile.TarInfo*), 605
isdisjoint() (*méthode frozenset*), 88
isdown() (*dans le module turtle*), 1586
iselement() (*dans le module xml.etree.ElementTree*), 1363
isenabled() (*dans le module gc*), 2022
isEnabledFor() (*méthode logging.Logger*), 759
isendwin() (*dans le module curses*), 927
ISEOF() (*dans le module token*), 2143
isfifo() (*méthode tarfile.TarInfo*), 605
isfile() (*dans le module os.path*), 475
isfile() (*méthode tarfile.TarInfo*), 605
isfinite() (*dans le module cmath*), 349
isfinite() (*dans le module math*), 342
isfirstline() (*dans le module fileinput*), 922
isframe() (*dans le module inspect*), 2029
isfunction() (*dans le module inspect*), 2028
isfuture() (*dans le module asyncio*), 1131
isgenerator() (*dans le module inspect*), 2028
isgeneratorfunction() (*dans le module inspect*), 2028
isgetsetdescriptor() (*dans le module inspect*), 2030
isgraph() (*dans le module curses.ascii*), 952
isidentifier() (*méthode str*), 57
isinf() (*dans le module cmath*), 349
isinf() (*dans le module math*), 342
isinstance()
    built-in function, 20
isjunction() (*dans le module os.path*), 475
iskeyword() (*dans le module keyword*), 2147
isleap() (*dans le module calendar*), 254
islice() (*dans le module itertools*), 416
islink() (*dans le module os.path*), 475
islnk() (*méthode tarfile.TarInfo*), 605
islower() (*dans le module curses.ascii*), 952
islower() (*méthode bytearray*), 75
islower() (*méthode bytes*), 75
islower() (*méthode str*), 57
ismemberdescriptor() (*dans le module inspect*), 2030
ismeta() (*dans le module curses.ascii*), 952
ismethod() (*dans le module inspect*), 2028
ismethoddescriptor() (*dans le module inspect*), 2030
ismethodwrapper() (*dans le module inspect*), 2030
ismodule() (*dans le module inspect*), 2028
ismount() (*dans le module os.path*), 475
isnan() (*dans le module cmath*), 349
isnan() (*dans le module math*), 342
ISNONTERMINAL() (*dans le module token*), 2143
IsNot (*classe dans ast*), 2111
isnumeric() (*méthode str*), 57
isocalendar() (*méthode datetime.date*), 216
isocalendar() (*méthode datetime.datetime*), 225
isoformat() (*méthode datetime.date*), 216
isoformat() (*méthode datetime.datetime*), 226
isoformat() (*méthode datetime.time*), 231

isolated (*attribut sys.flags*), 1936
IsolatedAsyncioTestCase (*classe dans unittest*), 1771
isolation_level (*attribut sqlite3.Connection*), 551
isoweekday() (*méthode datetime.date*), 216
isoweekday() (*méthode datetime.datetime*), 225
ispackage() (*dans le module inspect*), 2028
isprint() (*dans le module curses.ascii*), 952
isprintable() (*méthode str*), 57
ispunct() (*dans le module curses.ascii*), 952
isqrt() (*dans le module math*), 340
isreadable() (*dans le module pprint*), 308
isreadable() (*méthode pprint.PrettyPrinter*), 309
isrecursive() (*dans le module pprint*), 308
isrecursive() (*méthode pprint.PrettyPrinter*), 309
isreg() (*méthode tarfile.TarInfo*), 605
isreserved() (*dans le module os.path*), 475
isReservedKey() (*méthode http.cookies.Morsel*), 1511
isroutine() (*dans le module inspect*), 2030
isSameNode() (*méthode xml.dom.Node*), 1377
issoftkeyword() (*dans le module keyword*), 2147
isspace() (*dans le module curses.ascii*), 952
isspace() (*méthode bytearray*), 75
isspace() (*méthode bytes*), 75
isspace() (*méthode str*), 57
isstdin() (*dans le module fileinput*), 922
issubclass()
built-in function, 20
issubset() (*méthode frozenset*), 89
issuperset() (*méthode frozenset*), 89
issym() (*méthode tarfile.TarInfo*), 605
ISTERMINAL() (*dans le module token*), 2143
istitle() (*méthode bytearray*), 76
istitle() (*méthode bytes*), 76
istitle() (*méthode str*), 57
istraceback() (*dans le module inspect*), 2029
isub() (*dans le module operator*), 444
isupper() (*dans le module curses.ascii*), 952
isupper() (*méthode bytearray*), 76
isupper() (*méthode bytes*), 76
isupper() (*méthode str*), 57
isvisible() (*dans le module turtle*), 1589
isxdigit() (*dans le module curses.ascii*), 952
ITALIC (*dans le module tkinter.font*), 1633
item() (*méthode tkinter.ttk.Treeview*), 1653
item() (*méthode xml.dom.NamedNodeMap*), 1381
item() (*méthode xml.dom.NodeList*), 1378
itemgetter() (*dans le module operator*), 441
items() (*méthode configparser.ConfigParser*), 636
items() (*méthode contextvars.Context*), 1055
items() (*méthode dict*), 92
items() (*méthode email.message.EmailMessage*), 1253
items() (*méthode email.message.Message*), 1291
items() (*méthode mailbox.Mailbox*), 1319
items() (*méthode types.MappingProxyType*), 304
items() (*méthode xml.etree.ElementTree.Element*), 1367
itemsize (*attribut array.array*), 290
itemsize (*attribut memoryview*), 87
ItemsView (*classe dans collections.abc*), 280
ItemsView (*classe dans typing*), 1721
iter()
built-in function, 21
iter() (*méthode xml.etree.ElementTree.Element*), 1367
iter() (*méthode xml.etree.ElementTree.ElementTree*), 1369
iter_attachments() (*méthode email.message.EmailMessage*), 1256
iter_child_nodes() (*dans le module ast*), 2136
iter_fields() (*dans le module ast*), 2136
iter_importers() (*dans le module pkgutil*), 2060
iter_modules() (*dans le module pkgutil*), 2060
iter_parts() (*méthode email.message.EmailMessage*), 1257
iter_unpack() (*dans le module struct*), 182
iter_unpack() (*méthode struct.Struct*), 188
itérable, **2237**
Iterable (*classe dans collections.abc*), 279
Iterable (*classe dans typing*), 1722
itérable asynchrone, **2230**
itérateur, **2237**
itérateur asynchrone, **2230**
itérateur de générateur, **2235**
itérateur de générateur asynchrone, **2230**
Iterator (*classe dans collections.abc*), 279
Iterator (*classe dans typing*), 1722
iterator protocol, 46
iterdecode() (*dans le module codecs*), 191
iterdir() (*méthode importlib.abc.Traversable*), 2073
iterdir() (*méthode importlib.resources.abc.Traversable*), 2089
iterdir() (*méthode pathlib.Path*), 463
iterdir() (*méthode zipfile.Path*), 591
iterdump() (*méthode sqlite3.Connection*), 548
iterencode() (*dans le module codecs*), 191
iterencode() (*méthode json.JSONEncoder*), 1314
iterfind() (*méthode xml.etree.ElementTree.Element*), 1367
iterfind() (*méthode xml.etree.ElementTree.ElementTree*), 1369
iteritems() (*méthode mailbox.Mailbox*), 1319
iterkeys() (*méthode mailbox.Mailbox*), 1318
itermonthdates() (*méthode calendar.Calendar*), 251
itermonthdays() (*méthode calendar.Calendar*), 251
itermonthdays2() (*méthode calendar.Calendar*), 251
itermonthdays3() (*méthode calendar.Calendar*), 251
itermonthdays4() (*méthode calendar.Calendar*), 251
iterparse() (*dans le module xml.etree.ElementTree*), 1363
itertext() (*méthode xml.etree.ElementTree.Element*), 1368
itertools
module, 409
itervalues() (*méthode mailbox.Mailbox*), 1318
iterweekdays() (*méthode calendar.Calendar*), 250

ITIMER_PROF (*dans le module signal*), 1237
ITIMER_REAL (*dans le module signal*), 1237
ITIMER_VIRTUAL (*dans le module signal*), 1237
ItimerError, 1238
itruediv() (*dans le module operator*), 444
ixor() (*dans le module operator*), 444

J

-j
    option de ligne de commande compileall, 2157
JANUARY (*dans le module calendar*), 256
java_ver() (*dans le module platform*), 802
join() (*dans le module os.path*), 476
join() (*dans le module shlex*), 1613
join() (*méthode asyncio.Queue*), 1103
join() (*méthode bytearray*), 70
join() (*méthode bytes*), 70
join() (*méthode multiprocessing.JoinableQueue*), 982
join() (*méthode multiprocessing.pool.Pool*), 1000
join() (*méthode multiprocessing.Process*), 977
join() (*méthode queue.Queue*), 1051
join() (*méthode str*), 58
join() (*méthode threading.Thread*), 960
join_thread() (*dans le module test.support.threading_helper*), 1858
join_thread() (*méthode multiprocessing.Queue*), 981
JoinableQueue (*classe dans multiprocessing*), 982
JoinedStr (*classe dans ast*), 2107
joinpath() (*méthode importlib.abc.Traversable*), 2073
joinpath() (*méthode importlib.resources.abc.Traversable*), 2089
joinpath() (*méthode pathlib.PurePath*), 454
joinpath() (*méthode zipfile.Path*), 591
js_output() (*méthode http.cookies.BaseCookie*), 1510
js_output() (*méthode http.cookies.Morsel*), 1511
json
    module, 1307
JSONDecodeError, 1314
JSONDecoder (*classe dans json*), 1312
JSONEncoder (*classe dans json*), 1312
--json-lines
    option de ligne de commande json, 1317
json.tool
    module, 1316
JULY (*dans le module calendar*), 256
JUMP (*monitoring event*), 1956
JUMP (*opcode*), 2180
jump (*pdb command*), 1882
JUMP_BACKWARD (*opcode*), 2174
JUMP_BACKWARD_NO_INTERRUPT (*opcode*), 2174
JUMP_FORWARD (*opcode*), 2174
JUMP_IF_FALSE (*opcode*), 2180
JUMP_IF_TRUE (*opcode*), 2180
JUMP_NO_INTERRUPT (*opcode*), 2180
JUNE (*dans le module calendar*), 256

K

-k
    option de ligne de commande unittest, 1754
kbhit() (*dans le module msvcrt*), 2184
kde() (*dans le module statistics*), 396
kde_random() (*dans le module statistics*), 397
KEEP (*attribut enum.FlagBoundary*), 328
kevent() (*dans le module select*), 1225
key (*attribut http.cookies.Morsel*), 1511
key (*attribut zoneinfo.ZoneInfo*), 248
KEY_A1 (*dans le module curses*), 940
KEY_A3 (*dans le module curses*), 941
KEY_ALL_ACCESS (*dans le module winreg*), 2192
KEY_B2 (*dans le module curses*), 941
KEY_BACKSPACE (*dans le module curses*), 939
KEY_BEG (*dans le module curses*), 941
KEY_BREAK (*dans le module curses*), 939
KEY_BTAB (*dans le module curses*), 941
KEY_C1 (*dans le module curses*), 941
KEY_C3 (*dans le module curses*), 941
KEY_CANCEL (*dans le module curses*), 941
KEY_CATAB (*dans le module curses*), 940
KEY_CLEAR (*dans le module curses*), 940
KEY_CLOSE (*dans le module curses*), 941
KEY_COMMAND (*dans le module curses*), 941
KEY_COPY (*dans le module curses*), 941
KEY_CREATE (*dans le module curses*), 941
KEY_CREATE_LINK (*dans le module winreg*), 2192
KEY_CREATE_SUB_KEY (*dans le module winreg*), 2192
KEY_CTAB (*dans le module curses*), 940
KEY_DC (*dans le module curses*), 939
KEY_DL (*dans le module curses*), 939
KEY_DOWN (*dans le module curses*), 939
KEY_EIC (*dans le module curses*), 940
KEY_END (*dans le module curses*), 941
KEY_ENTER (*dans le module curses*), 940
KEY_ENUMERATE_SUB_KEYS (*dans le module winreg*), 2192
KEY_EOL (*dans le module curses*), 940
KEY_EOS (*dans le module curses*), 940
KEY_EXECUTE (*dans le module winreg*), 2192
KEY_EXIT (*dans le module curses*), 941
KEY_F0 (*dans le module curses*), 939
KEY_FIND (*dans le module curses*), 941
KEY_Fn (*dans le module curses*), 939
KEY_HELP (*dans le module curses*), 941
KEY_HOME (*dans le module curses*), 939
KEY_IC (*dans le module curses*), 940
KEY_IL (*dans le module curses*), 939
KEY_LEFT (*dans le module curses*), 939
KEY_LL (*dans le module curses*), 940
KEY_MARK (*dans le module curses*), 941
KEY_MAX (*dans le module curses*), 944
KEY_MESSAGE (*dans le module curses*), 941
KEY_MIN (*dans le module curses*), 939
KEY_MOUSE (*dans le module curses*), 944
KEY_MOVE (*dans le module curses*), 941

KEY_NEXT (*dans le module curses*), 942
KEY_NOTIFY (*dans le module winreg*), 2192
KEY_NPAGE (*dans le module curses*), 940
KEY_OPEN (*dans le module curses*), 942
KEY_OPTIONS (*dans le module curses*), 942
KEY_PPAGE (*dans le module curses*), 940
KEY_PREVIOUS (*dans le module curses*), 942
KEY_PRINT (*dans le module curses*), 940
KEY_QUERY_VALUE (*dans le module winreg*), 2192
KEY_READ (*dans le module winreg*), 2192
KEY_REDO (*dans le module curses*), 942
KEY_REFERENCE (*dans le module curses*), 942
KEY_REFRESH (*dans le module curses*), 942
KEY_REPLACE (*dans le module curses*), 942
KEY_RESET (*dans le module curses*), 940
KEY_RESIZE (*dans le module curses*), 944
KEY_RESTART (*dans le module curses*), 942
KEY_RESUME (*dans le module curses*), 942
KEY_RIGHT (*dans le module curses*), 939
KEY_SAVE (*dans le module curses*), 942
KEY_SBEG (*dans le module curses*), 942
KEY_SCANCEL (*dans le module curses*), 942
KEY_SCOMMAND (*dans le module curses*), 942
KEY_SCOPY (*dans le module curses*), 942
KEY_SCREATE (*dans le module curses*), 942
KEY_SDC (*dans le module curses*), 942
KEY_SDL (*dans le module curses*), 942
KEY_SELECT (*dans le module curses*), 943
KEY_SEND (*dans le module curses*), 943
KEY_SEOL (*dans le module curses*), 943
KEY_SET_VALUE (*dans le module winreg*), 2192
KEY_SEXIT (*dans le module curses*), 943
KEY_SF (*dans le module curses*), 940
KEY_SFIND (*dans le module curses*), 943
KEY_SHELP (*dans le module curses*), 943
KEY_SHOME (*dans le module curses*), 943
KEY_SIC (*dans le module curses*), 943
KEY_SLEFT (*dans le module curses*), 943
KEY_SMESSAGE (*dans le module curses*), 943
KEY_SMOVE (*dans le module curses*), 943
KEY_SNEXT (*dans le module curses*), 943
KEY_SOPTIONS (*dans le module curses*), 943
KEY_SPREVIOUS (*dans le module curses*), 943
KEY_SPRINT (*dans le module curses*), 943
KEY_SR (*dans le module curses*), 940
KEY_SREDO (*dans le module curses*), 943
KEY_SREPLACE (*dans le module curses*), 943
KEY_SRESET (*dans le module curses*), 940
KEY_SRIGHT (*dans le module curses*), 943
KEY_SRSUME (*dans le module curses*), 944
KEY_SSAVE (*dans le module curses*), 944
KEY_SSUSPEND (*dans le module curses*), 944
KEY_STAB (*dans le module curses*), 940
KEY_SUNDO (*dans le module curses*), 944
KEY_SUSPEND (*dans le module curses*), 944
KEY_UNDO (*dans le module curses*), 944
KEY_UP (*dans le module curses*), 939
KEY_WOW64_32KEY (*dans le module winreg*), 2192
KEY_WOW64_64KEY (*dans le module winreg*), 2192
KEY_WRITE (*dans le module winreg*), 2192
KeyboardInterrupt, 110
KeyError, 110
keylog_filename (*attribut ssl.SSLContext*), 1212
keyname() (*dans le module curses*), 927
keypad() (*méthode curses.window*), 935
keyrefs() (*méthode weakref.WeakKeyDictionary*), 294
keys() (*méthode contextvars.Context*), 1055
keys() (*méthode dict*), 92
keys() (*méthode email.message.EmailMessage*), 1253
keys() (*méthode email.message.Message*), 1291
keys() (*méthode mailbox.Mailbox*), 1318
keys() (*méthode sqlite3.Row*), 555
keys() (*méthode types.MappingProxyType*), 304
keys() (*méthode xml.etree.ElementTree.Element*), 1367
KeysView (*classe dans collections.abc*), 280
KeysView (*classe dans typing*), 1721
keyword
    module, 2147
keyword (*classe dans ast*), 2112
keywords (*attribut functools.partial*), 437
kill() (*dans le module os*), 716
kill() (*méthode asyncio.subprocess.Process*), 1100
kill() (*méthode asyncio.SubprocessTransport*), 1138
kill() (*méthode multiprocessing.Process*), 978
kill() (*méthode subprocess.Popen*), 1038
kill_python() (*dans le module test.support.script_helper*), 1857
killchar() (*dans le module curses*), 927
killpg() (*dans le module os*), 716
kind (*attribut inspect.Parameter*), 2034
knownfiles (*dans le module mimetypes*), 1337
kqueue() (*dans le module select*), 1225
KqueueSelector (*classe dans selectors*), 1233
KW_ONLY (*dans le module dataclasses*), 1984
kwargs (*attribut inspect.BoundArguments*), 2036
kwargs (*attribut typing.ParamSpec*), 1700
kwlist (*dans le module keyword*), 2147

L

-L
    option de ligne de commande calendar, 257
-l
    option de ligne de commande calendar, 258
    option de ligne de commande compileall, 2156
    option de ligne de commande pickletools, 2182
    option de ligne de commande tarfile, 609
    option de ligne de commande trace, 1899
    option de ligne de commande zipfile, 594
L (*dans le module re*), 141

lambda, **2238**
Lambda (*classe dans ast*), 2130
LambdaType (*dans le module types*), 301
LANG, 1553, 1555, 1561, 1565
LANGUAGE, 1553, 1554
language
    C, 38
large files, 2197
LARGEST (*dans le module test.support*), 1849
LargeZipFile, 584
last_accepted (*attribut multiprocessing.connection.Listener*), 1002
last_exc (*dans le module sys*), 1945
last_traceback (*dans le module sys*), 1945
last_type (*dans le module sys*), 1945
last_value (*dans le module sys*), 1945
lastChild (*attribut xml.dom.Node*), 1377
lastcmd (*attribut cmd.Cmd*), 1610
lastgroup (*attribut re.Match*), 150
lastindex (*attribut re.Match*), 150
lastResort (*dans le module logging*), 773
lastrowid (*attribut sqlite3.Cursor*), 554
layout() (*méthode tkinter.ttk.Style*), 1656
lazycache() (*dans le module linecache*), 497
LazyLoader (*classe dans importlib.util*), 2081
LBRACE (*dans le module token*), 2145
LBYL, **2238**
LC_ALL, 1553, 1554
LC_ALL (*dans le module locale*), 1567
LC_COLLATE (*dans le module locale*), 1567
LC_CTYPE (*dans le module locale*), 1567
LC_MESSAGES, 1553, 1555
LC_MESSAGES (*dans le module locale*), 1567
LC_MONETARY (*dans le module locale*), 1567
LC_NUMERIC (*dans le module locale*), 1567
LC_TIME (*dans le module locale*), 1567
lchflags() (*dans le module os*), 689
lchmod() (*dans le module os*), 689
lchmod() (*méthode pathlib.Path*), 469
lchown() (*dans le module os*), 689
lcm() (*dans le module math*), 340
ldexp() (*dans le module math*), 342
Le zen de Python, **2245**
le() (*dans le module operator*), 437
leapdays() (*dans le module calendar*), 254
leaveok() (*méthode curses.window*), 935
left (*attribut filecmp.dircmp*), 486
left() (*dans le module turtle*), 1579
left_list (*attribut filecmp.dircmp*), 486
left_only (*attribut filecmp.dircmp*), 486
LEFTSHIFT (*dans le module token*), 2145
LEFTSHIFTEQUAL (*dans le module token*), 2145
LEGACY_TRANSACTION_CONTROL (*dans le module sqlite3*), 540
len
    built-in function, 47, 90
len()
    built-in function, 21
length (*attribut xml.dom.NamedNodeMap*), 1381
length (*attribut xml.dom.NodeList*), 1378
length_hint() (*dans le module operator*), 440
LESS (*dans le module token*), 2144
LESSEQUAL (*dans le module token*), 2145
level (*attribut logging.Logger*), 757
LexicalHandler (*classe dans xml.sax.handler*), 1393
lexists() (*dans le module os.path*), 474
LF (*dans le module curses.ascii*), 950
lgamma() (*dans le module math*), 346
libc_ver() (*dans le module platform*), 803
LIBRARIES_ASSEMBLY_NAME_PREFIX (*dans le module msvcrt*), 2185
library (*attribut ssl.SSLError*), 1192
library (*dans le module dbm.ndbm*), 534
LibraryLoader (*classe dans ctypes*), 835
license (*variable de base*), 36
LifoQueue (*classe dans asyncio*), 1103
LifoQueue (*classe dans queue*), 1049
light-weight processes, 1056
limit_denominator() (*méthode fractions.Fraction*), 382
LimitOverrunError, 1105
line (*attribut bdb.Breakpoint*), 1870
line (*attribut traceback.FrameSummary*), 2016
LINE (*monitoring event*), 1956
line_buffering (*attribut io.TextIOWrapper*), 741
line_num (*attribut csv.csvreader*), 618
linear_regression() (*dans le module statistics*), 403
line-buffered I/O, 25
linecache
    module, 496
lineno (*attribut ast.AST*), 2105
lineno (*attribut doctest.DocTest*), 1744
lineno (*attribut doctest.Example*), 1744
lineno (*attribut inspect.FrameInfo*), 2039
lineno (*attribut inspect.Traceback*), 2039
lineno (*attribut json.JSONDecodeError*), 1314
lineno (*attribut netrc.NetrcParseError*), 641
lineno (*attribut pyclbr.Class*), 2153
lineno (*attribut pyclbr.Function*), 2153
lineno (*attribut re.PatternError*), 146
lineno (*attribut shlex.shlex*), 1616
lineno (*attribut SyntaxError*), 112
lineno (*attribut tomllib.TOMLDecodeError*), 640
lineno (*attribut traceback.FrameSummary*), 2016
lineno (*attribut traceback.TracebackException*), 2014
lineno (*attribut tracemalloc.Filter*), 1907
lineno (*attribut tracemalloc.Frame*), 1908
lineno (*attribut xml.parsers.expat.ExpatError*), 1408
lineno() (*dans le module fileinput*), 922
LINES, 925, 930
--lines
    option de ligne de commande calendar, 258
lines (*attribut os.terminal_size*), 685
LINES (*dans le module curses*), 937
linesep (*attribut email.policy.Policy*), 1266

linesep (*dans le module os*), 728
lineterminator (*attribut csv.Dialect*), 618
LineTooLong, 1459
link() (*dans le module os*), 690
linkname (*attribut tarfile.TarInfo*), 604
LinkOutsideDestinationError, 598
list
    object, 49
    type, operations on, 49
--list
    option de ligne de commande tarfile, 609
    option de ligne de commande zipfile, 594
List (*classe dans ast*), 2108
List (*classe dans typing*), 1719
list (*classe de base*), 49
list (*pdb command*), 1882
list() (*méthode imaplib.IMAP4*), 1478
list() (*méthode multiprocessing.managers.SyncManager*), 993
list() (*méthode poplib.POP3*), 1473
list() (*méthode tarfile.TarFile*), 600
LIST_APPEND (*opcode*), 2169
list_dialects() (*dans le module csv*), 615
LIST_EXTEND (*opcode*), 2172
list_folders() (*méthode mailbox.Maildir*), 1321
list_folders() (*méthode mailbox.MH*), 1324
ListComp (*classe dans ast*), 2113
listdir() (*dans le module os*), 690
listdrives() (*dans le module os*), 690
liste, **2238**
liste en compréhension (*ou liste en intension*), **2238**
listen() (*dans le module logging.config*), 775
listen() (*dans le module turtle*), 1597
listen() (*méthode socket.socket*), 1180
listener (*attribut logging.handlers.QueueHandler*), 798
Listener (*classe dans multiprocessing.connection*), 1001
--listfuncs
    option de ligne de commande trace, 1899
listMethods() (*méthode xmlrpc.client.ServerProxy.system*), 1523
listmounts() (*dans le module os*), 691
listvolumes() (*dans le module os*), 691
listxattr() (*dans le module os*), 711
Literal (*dans le module typing*), 1688
literal_eval() (*dans le module ast*), 2135
literals
    binary, 39
    complex number, 39
    floating-point, 39
    hexadecimal, 39
    integer, 39
    numeric, 39
    octal, 39
LiteralString (*dans le module typing*), 1684
LittleEndianStructure (*classe dans ctypes*), 846
LittleEndianUnion (*classe dans ctypes*), 846
ljust() (*méthode bytearray*), 71
ljust() (*méthode bytes*), 71
ljust() (*méthode str*), 58
LK_LOCK (*dans le module msvcrt*), 2183
LK_NBLCK (*dans le module msvcrt*), 2183
LK_NBRLCK (*dans le module msvcrt*), 2183
LK_RLCK (*dans le module msvcrt*), 2183
LK_UNLCK (*dans le module msvcrt*), 2183
ll (*pdb command*), 1882
LMTP (*classe dans smtplib*), 1483
ln() (*méthode decimal.Context*), 367
ln() (*méthode decimal.Decimal*), 359
LNKTYPE (*dans le module tarfile*), 598
Load (*classe dans ast*), 2109
load() (*dans le module json*), 1311
load() (*dans le module marshal*), 530
load() (*dans le module pickle*), 511
load() (*dans le module plistlib*), 642
load() (*dans le module tomllib*), 639
load() (*méthode de la classe tracemalloc.Snapshot*), 1908
load() (*méthode http.cookiejar.FileCookieJar*), 1515
load() (*méthode http.cookies.BaseCookie*), 1510
load() (*méthode pickle.Unpickler*), 514
LOAD_ATTR (*opcode*), 2173
LOAD_BUILD_CLASS (*opcode*), 2170
load_cert_chain() (*méthode ssl.SSLContext*), 1207
LOAD_CLOSURE (*opcode*), 2180
LOAD_COMMON_CONSTANT (*opcode*), 2170
LOAD_CONST (*opcode*), 2171
LOAD_CONST_IMMORTAL (*opcode*), 2171
load_default_certs() (*méthode ssl.SSLContext*), 1207
LOAD_DEREF (*opcode*), 2175
load_dh_params() (*méthode ssl.SSLContext*), 1210
load_extension() (*méthode sqlite3.Connection*), 548
LOAD_FAST (*opcode*), 2174
LOAD_FAST_AND_CLEAR (*opcode*), 2175
LOAD_FAST_CHECK (*opcode*), 2175
LOAD_FAST_LOAD_FAST (*opcode*), 2175
LOAD_FROM_DICT_OR_DEREF (*opcode*), 2175
LOAD_FROM_DICT_OR_GLOBALS (*opcode*), 2172
LOAD_GLOBAL (*opcode*), 2174
LOAD_LOCALS (*opcode*), 2172
LOAD_METHOD (*opcode*), 2180
load_module() (*méthode importlib.abc.FileLoader*), 2071
load_module() (*méthode importlib.abc.InspectLoader*), 2071
load_module() (*méthode importlib.abc.Loader*), 2069
load_module() (*méthode importlib.abc.SourceLoader*), 2072
load_module() (*méthode importlib.machinery.SourceFileLoader*), 2076

load_module() (*méthode importlib.machinery.SourcelessFileLoader*), 2077
load_module() (*méthode zipimport.zipimporter*), 2059
LOAD_NAME (*opcode*), 2171
load_package_tests() (*dans le module test.support*), 1854
LOAD_SMALL_INT (*opcode*), 2171
LOAD_SPECIAL (*opcode*), 2179
LOAD_SUPER_ATTR (*opcode*), 2173
load_verify_locations() (*méthode ssl.SSLContext*), 1207
loader (*attribut importlib.machinery.ModuleSpec*), 2078
Loader (*classe dans importlib.abc*), 2069
loader_state (*attribut importlib.machinery.ModuleSpec*), 2078
LoadError, 1513
LoadFileDialog (*classe dans tkinter.filedialog*), 1636
LoadKey() (*dans le module winreg*), 2188
LoadLibrary() (*méthode ctypes.LibraryLoader*), 835
loads() (*dans le module json*), 1311
loads() (*dans le module marshal*), 530
loads() (*dans le module pickle*), 512
loads() (*dans le module plistlib*), 643
loads() (*dans le module tomllib*), 640
loads() (*dans le module xmlrpc.client*), 1527
loadTestsFromModule() (*méthode unittest.TestLoader*), 1774
loadTestsFromName() (*méthode unittest.TestLoader*), 1774
loadTestsFromNames() (*méthode unittest.TestLoader*), 1775
loadTestsFromTestCase() (*méthode unittest.TestLoader*), 1774
local (*classe dans threading*), 958
LOCAL_CREDS (*dans le module socket*), 1170
LOCAL_CREDS_PERSISTENT (*dans le module socket*), 1170
localcontext() (*dans le module decimal*), 363
locale
    module, 1561
--locale
    option de ligne de commande calendar, 257
LOCALE (*dans le module re*), 141
localeconv() (*dans le module locale*), 1561
LocaleHTMLCalendar (*classe dans calendar*), 254
LocaleTextCalendar (*classe dans calendar*), 254
localize() (*dans le module locale*), 1567
localName (*attribut xml.dom.Attr*), 1381
localName (*attribut xml.dom.Node*), 1377
--locals
    option de ligne de commande unittest, 1754
locals()
    built-in function, 21
localtime() (*dans le module email.utils*), 1304
localtime() (*dans le module time*), 746
Locator (*classe dans xml.sax.xmlreader*), 1399
lock (*attribut sys.thread_info*), 1953
Lock (*classe dans asyncio*), 1092
Lock (*classe dans multiprocessing*), 986
Lock (*classe dans threading*), 961
lock() (*méthode mailbox.Babyl*), 1326
lock() (*méthode mailbox.Mailbox*), 1320
lock() (*méthode mailbox.Maildir*), 1322
lock() (*méthode mailbox.mbox*), 1323
lock() (*méthode mailbox.MH*), 1324
lock() (*méthode mailbox.MMDF*), 1326
Lock() (*méthode multiprocessing.managers.SyncManager*), 993
LOCK_EX (*dans le module fcntl*), 2206
LOCK_NB (*dans le module fcntl*), 2206
LOCK_SH (*dans le module fcntl*), 2205
LOCK_UN (*dans le module fcntl*), 2205
locked() (*méthode _thread.lock*), 1058
locked() (*méthode asyncio.Condition*), 1094
locked() (*méthode asyncio.Lock*), 1093
locked() (*méthode asyncio.Semaphore*), 1096
locked() (*méthode threading.Lock*), 962
lockf() (*dans le module fcntl*), 2205
lockf() (*dans le module os*), 676
locking() (*dans le module msvcrt*), 2183
LockType (*dans le module _thread*), 1057
log() (*dans le module cmath*), 348
log() (*dans le module logging*), 770
log() (*dans le module math*), 343
log() (*méthode logging.Logger*), 761
log1p() (*dans le module math*), 343
log2() (*dans le module math*), 343
log10() (*dans le module cmath*), 348
log10() (*dans le module math*), 344
log10() (*méthode decimal.Context*), 367
log10() (*méthode decimal.Decimal*), 360
LOG_ALERT (*dans le module syslog*), 2211
LOG_AUTH (*dans le module syslog*), 2212
LOG_AUTHPRIV (*dans le module syslog*), 2212
LOG_CONS (*dans le module syslog*), 2212
LOG_CRIT (*dans le module syslog*), 2211
LOG_CRON (*dans le module syslog*), 2212
LOG_DAEMON (*dans le module syslog*), 2212
log_date_time_string() (*méthode http.server.BaseHTTPRequestHandler*), 1506
LOG_DEBUG (*dans le module syslog*), 2211
LOG_EMERG (*dans le module syslog*), 2211
LOG_ERR (*dans le module syslog*), 2211
log_error() (*méthode http.server.BaseHTTPRequestHandler*), 1505
log_exception() (*méthode wsgiref.handlers.BaseHandler*), 1423
LOG_FTP (*dans le module syslog*), 2212
LOG_INFO (*dans le module syslog*), 2211
LOG_INSTALL (*dans le module syslog*), 2212
LOG_KERN (*dans le module syslog*), 2212

LOG_LAUNCHD (*dans le module syslog*), 2212
LOG_LOCAL0 (*dans le module syslog*), 2212
LOG_LOCAL1 (*dans le module syslog*), 2212
LOG_LOCAL2 (*dans le module syslog*), 2212
LOG_LOCAL3 (*dans le module syslog*), 2212
LOG_LOCAL4 (*dans le module syslog*), 2212
LOG_LOCAL5 (*dans le module syslog*), 2212
LOG_LOCAL6 (*dans le module syslog*), 2212
LOG_LOCAL7 (*dans le module syslog*), 2212
LOG_LPR (*dans le module syslog*), 2212
LOG_MAIL (*dans le module syslog*), 2212
log_message() (*méthode http.server.BaseHTTPRequestHandler*), 1505
LOG_NDELAY (*dans le module syslog*), 2212
LOG_NETINFO (*dans le module syslog*), 2212
LOG_NEWS (*dans le module syslog*), 2212
LOG_NOTICE (*dans le module syslog*), 2211
LOG_NOWAIT (*dans le module syslog*), 2212
LOG_ODELAY (*dans le module syslog*), 2212
LOG_PERROR (*dans le module syslog*), 2212
LOG_PID (*dans le module syslog*), 2212
LOG_RAS (*dans le module syslog*), 2212
LOG_REMOTEAUTH (*dans le module syslog*), 2212
log_request() (*méthode http.server.BaseHTTPRequestHandler*), 1505
LOG_SYSLOG (*dans le module syslog*), 2212
log_to_stderr() (*dans le module multiprocessing*), 1004
LOG_USER (*dans le module syslog*), 2212
LOG_UUCP (*dans le module syslog*), 2212
LOG_WARNING (*dans le module syslog*), 2211
logb() (*méthode decimal.Context*), 367
logb() (*méthode decimal.Decimal*), 360
Logger (*classe dans logging*), 757
LoggerAdapter (*classe dans logging*), 769
logging
    Errors, 756
    module, 756
logging.config
    module, 774
logging.handlers
    module, 786
logical_and() (*méthode decimal.Context*), 367
logical_and() (*méthode decimal.Decimal*), 360
logical_invert() (*méthode decimal.Context*), 367
logical_invert() (*méthode decimal.Decimal*), 360
logical_or() (*méthode decimal.Context*), 367
logical_or() (*méthode decimal.Decimal*), 360
logical_xor() (*méthode decimal.Context*), 367
logical_xor() (*méthode decimal.Decimal*), 360
login() (*méthode ftplib.FTP*), 1466
login() (*méthode imaplib.IMAP4*), 1478
login() (*méthode smtplib.SMTP*), 1485
login_cram_md5() (*méthode imaplib.IMAP4*), 1478
login_tty() (*dans le module os*), 676
LOGNAME, 668, 921
lognormvariate() (*dans le module random*), 387
logout() (*méthode imaplib.IMAP4*), 1478
LogRecord (*classe dans logging*), 766
LONG_TIMEOUT (*dans le module test.support*), 1848
longMessage (*attribut unittest.TestCase*), 1770
longname() (*dans le module curses*), 927
lookup() (*dans le module codecs*), 189
lookup() (*dans le module unicodedata*), 172
lookup() (*méthode symtable.SymbolTable*), 2140
lookup() (*méthode tkinter.ttk.Style*), 1655
lookup_error() (*dans le module codecs*), 193
LookupError, 109
loop
    over mutable sequence, 47
loop_factory (*attribut unittest.IsolatedAsyncioTestCase*), 1771
LOOPBACK_TIMEOUT (*dans le module test.support*), 1847
lower() (*méthode bytearray*), 76
lower() (*méthode bytes*), 76
lower() (*méthode str*), 58
LPAR (*dans le module token*), 2144
lpAttributeList (*attribut subprocess.STARTUPINFO*), 1040
lru_cache() (*dans le module functools*), 428
lseek() (*dans le module os*), 676
LShift (*classe dans ast*), 2110
lshift() (*dans le module operator*), 439
LSQB (*dans le module token*), 2144
lstat() (*dans le module os*), 691
lstat() (*méthode pathlib.Path*), 460
lstrip() (*méthode bytearray*), 72
lstrip() (*méthode bytes*), 72
lstrip() (*méthode str*), 58
lsub() (*méthode imaplib.IMAP4*), 1478
Lt (*classe dans ast*), 2111
lt() (*dans le module operator*), 437
lt() (*dans le module turtle*), 1579
LtE (*classe dans ast*), 2111
LWPCookieJar (*classe dans http.cookiejar*), 1516
lzma
    module, 579
LZMACompressor (*classe dans lzma*), 580
LZMADecompressor (*classe dans lzma*), 581
LZMAError, 579
LZMAFile (*classe dans lzma*), 579

M

-m
    option de ligne de commande ast, 2138
    option de ligne de commande calendar, 258
    option de ligne de commande pickletools, 2182
    option de ligne de commande trace, 1899
    option de ligne de commande zipapp, 1925
M (*dans le module re*), 141

mac_ver() (*dans le module platform*), 803
machine virtuelle, **2245**
machine() (*dans le module platform*), 800
macros (*attribut netrc.netrc*), 642
MADV_AUTOSYNC (*dans le module mmap*), 1247
MADV_CORE (*dans le module mmap*), 1247
MADV_DODUMP (*dans le module mmap*), 1247
MADV_DOFORK (*dans le module mmap*), 1247
MADV_DONTDUMP (*dans le module mmap*), 1247
MADV_DONTFORK (*dans le module mmap*), 1247
MADV_DONTNEED (*dans le module mmap*), 1247
MADV_FREE (*dans le module mmap*), 1247
MADV_FREE_REUSABLE (*dans le module mmap*), 1247
MADV_FREE_REUSE (*dans le module mmap*), 1247
MADV_HUGEPAGE (*dans le module mmap*), 1247
MADV_HWPOISON (*dans le module mmap*), 1247
MADV_MERGEABLE (*dans le module mmap*), 1247
MADV_NOCORE (*dans le module mmap*), 1247
MADV_NOHUGEPAGE (*dans le module mmap*), 1247
MADV_NORMAL (*dans le module mmap*), 1247
MADV_NOSYNC (*dans le module mmap*), 1247
MADV_PROTECT (*dans le module mmap*), 1247
MADV_RANDOM (*dans le module mmap*), 1247
MADV_REMOVE (*dans le module mmap*), 1247
MADV_SEQUENTIAL (*dans le module mmap*), 1247
MADV_SOFT_OFFLINE (*dans le module mmap*), 1247
MADV_UNMERGEABLE (*dans le module mmap*), 1247
MADV_WILLNEED (*dans le module mmap*), 1247
madvise() (*méthode mmap.mmap*), 1246
magic
    méthode, 2238
MAGIC_NUMBER (*dans le module importlib.util*), 2079
MagicMock (*classe dans unittest.mock*), 1813
mailbox
    module, 1317
Mailbox (*classe dans mailbox*), 1317
mailcap
    module, 2223
Maildir (*classe dans mailbox*), 1320
MaildirMessage (*classe dans mailbox*), 1327
--main
    option de ligne de commande zipapp, 1925
main() (*dans le module site*), 2051
main() (*dans le module unittest*), 1779
main_thread() (*dans le module threading*), 957
mainloop() (*dans le module turtle*), 1599
maintype (*attribut email.headerregistry.ContentTypeHeader*), 1276
major (*attribut email.headerregistry.MIMEVersionHeader*), 1276
major() (*dans le module os*), 693
make_alternative() (*méthode email.message.EmailMessage*), 1257
make_archive() (*dans le module shutil*), 503
make_bad_fd() (*dans le module test.support.os_helper*), 1860
MAKE_CELL (*opcode*), 2175
make_cookies() (*méthode http.cookiejar.CookieJar*), 1514
make_dataclass() (*dans le module dataclasses*), 1983
make_file() (*méthode difflib.HtmlDiff*), 157
MAKE_FUNCTION (*opcode*), 2177
make_header() (*dans le module email.header*), 1301
make_legacy_pyc() (*dans le module test.support.import_helper*), 1862
make_mixed() (*méthode email.message.EmailMessage*), 1257
make_msgid() (*dans le module email.utils*), 1304
make_parser() (*dans le module xml.sax*), 1391
make_pkg() (*dans le module test.support.script_helper*), 1857
make_related() (*méthode email.message.EmailMessage*), 1257
make_script() (*dans le module test.support.script_helper*), 1857
make_server() (*dans le module wsgiref.simple_server*), 1419
make_table() (*méthode difflib.HtmlDiff*), 157
make_zip_pkg() (*dans le module test.support.script_helper*), 1858
make_zip_script() (*dans le module test.support.script_helper*), 1857
makedev() (*dans le module os*), 693
makedirs() (*dans le module os*), 692
makeelement() (*méthode xml.etree.ElementTree.Element*), 1368
makefile() (*méthode socket.socket*), 1180
makeLogRecord() (*dans le module logging*), 771
makePickle() (*méthode logging.handlers.SocketHandler*), 792
makeRecord() (*méthode logging.Logger*), 761
makeSocket() (*méthode logging.handlers.DatagramHandler*), 793
makeSocket() (*méthode logging.handlers.SocketHandler*), 792
maketrans() (*méthode statique bytearray*), 70
maketrans() (*méthode statique bytes*), 70
maketrans() (*méthode statique str*), 58
manager (*attribut logging.LoggerAdapter*), 769
mangle_from_ (*attribut email.policy.Compat32*), 1271
mangle_from_ (*attribut email.policy.Policy*), 1267
MANPAGER, 1725
mant_dig (*attribut sys.float_info*), 1938
map()
    built-in function, 22
map() (*méthode concurrent.futures.Executor*), 1020
map() (*méthode multiprocessing.pool.Pool*), 999
map() (*méthode tkinter.ttk.Style*), 1655
MAP_32BIT (*dans le module mmap*), 1248
MAP_ADD (*opcode*), 2169
MAP_ALIGNED_SUPER (*dans le module mmap*), 1248
MAP_ANON (*dans le module mmap*), 1248
MAP_ANONYMOUS (*dans le module mmap*), 1248
map_async() (*méthode multiprocessing.pool.Pool*), 999
MAP_CONCEAL (*dans le module mmap*), 1248

MAP_DENYWRITE (*dans le module mmap*), 1248
MAP_EXECUTABLE (*dans le module mmap*), 1248
MAP_HASSEMAPHORE (*dans le module mmap*), 1248
MAP_JIT (*dans le module mmap*), 1248
MAP_NOCACHE (*dans le module mmap*), 1248
MAP_NOEXTEND (*dans le module mmap*), 1248
MAP_NORESERVE (*dans le module mmap*), 1248
MAP_POPULATE (*dans le module mmap*), 1248
MAP_PRIVATE (*dans le module mmap*), 1248
MAP_RESILIENT_CODESIGN (*dans le module mmap*), 1248
MAP_RESILIENT_MEDIA (*dans le module mmap*), 1248
MAP_SHARED (*dans le module mmap*), 1248
MAP_STACK (*dans le module mmap*), 1248
map_table_b2() (*dans le module stringprep*), 174
map_table_b3() (*dans le module stringprep*), 174
map_to_type() (*méthode email.headerregistry.HeaderRegistry*), 1277
MAP_TPRO (*dans le module mmap*), 1248
MAP_TRANSLATED_ALLOW_EXECUTE (*dans le module mmap*), 1248
MAP_UNIX03 (*dans le module mmap*), 1248
mapLogRecord() (*méthode logging.handlers.HTTPHandler*), 797
mapping
    object, 90
    types, operations on, 90
Mapping (*classe dans collections.abc*), 279
Mapping (*classe dans typing*), 1721
MappingProxyType (*classe dans types*), 304
MappingView (*classe dans collections.abc*), 280
MappingView (*classe dans typing*), 1721
mapPriority() (*méthode logging.handlers.SysLogHandler*), 795
maps (*attribut collections.ChainMap*), 259
MARCH (*dans le module calendar*), 256
markcoroutinefunction() (*dans le module inspect*), 2028
marshal
    module, 529
marshalling
    objects, 509
masking
    operations, 40
master (*attribut tkinter.Tk*), 1621
Match (*classe dans ast*), 2123
Match (*classe dans re*), 147
Match (*classe dans typing*), 1720
match() (*dans le module re*), 142
match() (*méthode pathlib.PurePath*), 454
match() (*méthode re.Pattern*), 146
match_case (*classe dans ast*), 2123
MATCH_CLASS (*opcode*), 2178
MATCH_KEYS (*opcode*), 2170
MATCH_MAPPING (*opcode*), 2170
MATCH_SEQUENCE (*opcode*), 2170
match_value() (*méthode test.support.Matcher*), 1856
MatchAs (*classe dans ast*), 2127
MatchClass (*classe dans ast*), 2126
Matcher (*classe dans test.support*), 1856
matches() (*méthode test.support.Matcher*), 1856
MatchMapping (*classe dans ast*), 2126
MatchOr (*classe dans ast*), 2128
MatchSequence (*classe dans ast*), 2125
MatchSingleton (*classe dans ast*), 2124
MatchStar (*classe dans ast*), 2125
MatchValue (*classe dans ast*), 2124
math
    module, 40, 338, 350
matmul() (*dans le module operator*), 439
MatMult (*classe dans ast*), 2110
max
    built-in function, 47
max (*attribut datetime.date*), 214
max (*attribut datetime.datetime*), 221
max (*attribut datetime.time*), 229
max (*attribut datetime.timedelta*), 210
max (*attribut sys.float_info*), 1938
MAX (*dans le module uuid*), 1492
max()
    built-in function, 22
max() (*méthode decimal.Context*), 367
max() (*méthode decimal.Decimal*), 360
max_10_exp (*attribut sys.float_info*), 1938
max_children (*attribut socketserver.ThreadingMixIn*), 1495
max_count (*attribut email.headerregistry.BaseHeader*), 1274
MAX_EMAX (*dans le module decimal*), 369
max_exp (*attribut sys.float_info*), 1938
MAX_INTERPOLATION_DEPTH (*dans le module configparser*), 637
max_line_length (*attribut email.policy.Policy*), 1266
max_lines (*attribut textwrap.TextWrapper*), 171
max_mag() (*méthode decimal.Context*), 368
max_mag() (*méthode decimal.Decimal*), 360
max_memuse (*dans le module test.support*), 1848
MAX_PREC (*dans le module decimal*), 369
max_prefixlen (*attribut ipaddress.IPv4Address*), 1535
max_prefixlen (*attribut ipaddress.IPv4Network*), 1540
max_prefixlen (*attribut ipaddress.IPv6Address*), 1538
max_prefixlen (*attribut ipaddress.IPv6Network*), 1543
MAX_Py_ssize_t (*dans le module test.support*), 1848
maxarray (*attribut reprlib.Repr*), 314
maxdeque (*attribut reprlib.Repr*), 314
maxdict (*attribut reprlib.Repr*), 314
maxDiff (*attribut unittest.TestCase*), 1770
maxfrozenset (*attribut reprlib.Repr*), 314
MAXIMUM_SUPPORTED (*attribut ssl.TLSVersion*), 1201
maximum_version (*attribut ssl.SSLContext*), 1212
maxlen (*attribut collections.deque*), 266
maxlevel (*attribut reprlib.Repr*), 314
maxlist (*attribut reprlib.Repr*), 314
maxlong (*attribut reprlib.Repr*), 314

maxother (*attribut reprlib.Repr*), 314
maxset (*attribut reprlib.Repr*), 314
maxsize (*attribut asyncio.Queue*), 1102
maxsize (*dans le module sys*), 1945
maxstring (*attribut reprlib.Repr*), 314
maxtuple (*attribut reprlib.Repr*), 314
maxunicode (*dans le module sys*), 1945
MAXYEAR (*dans le module datetime*), 208
MAY (*dans le module calendar*), 256
MB_ICONASTERISK (*dans le module winsound*), 2195
MB_ICONEXCLAMATION (*dans le module winsound*), 2196
MB_ICONHAND (*dans le module winsound*), 2196
MB_ICONQUESTION (*dans le module winsound*), 2196
MB_OK (*dans le module winsound*), 2196
mbox (*classe dans mailbox*), 1323
mboxMessage (*classe dans mailbox*), 1329
md5() (*dans le module hashlib*), 646
mean (*attribut statistics.NormalDist*), 404
mean() (*dans le module statistics*), 395
measure() (*méthode tkinter.font.Font*), 1633
median (*attribut statistics.NormalDist*), 404
median() (*dans le module statistics*), 397
median_grouped() (*dans le module statistics*), 398
median_high() (*dans le module statistics*), 398
median_low() (*dans le module statistics*), 398
member() (*dans le module enum*), 330
member_names (*attribut enum.EnumDict*), 328
MemberDescriptorType (*dans le module types*), 303
memfd_create() (*dans le module os*), 706
memmove() (*dans le module ctypes*), 841
--memo
    option de ligne de commande pickletools, 2182
MemoryBIO (*classe dans ssl*), 1221
MemoryError, 110
MemoryHandler (*classe dans logging.handlers*), 797
memoryview
    object, 65
memoryview (*classe de base*), 81
memoryview_at() (*dans le module ctypes*), 842
memset() (*dans le module ctypes*), 841
merge() (*dans le module heapq*), 282
message (*attribut BaseExceptionGroup*), 117
Message (*classe dans email.message*), 1288
Message (*classe dans mailbox*), 1326
Message (*classe dans tkinter.messagebox*), 1637
message digest, MD5, 645
message_factory (*attribut email.policy.Policy*), 1267
message_from_binary_file() (*dans le module email*), 1261
message_from_bytes() (*dans le module email*), 1261
message_from_file() (*dans le module email*), 1261
message_from_string() (*dans le module email*), 1261
MessageBeep() (*dans le module winsound*), 2194
MessageClass (*attribut http.server.BaseHTTPRequestHandler*), 1504
MessageDefect, 1272
MessageError, 1272
MessageParseError, 1272
messages (*dans le module xml.parsers.expat.errors*), 1410
meta() (*dans le module curses*), 927
meta_path (*dans le module sys*), 1945
métaclasse, **2238**
metadata() (*dans le module importlib.metadata*), 2092
--metadata-encoding
    option de ligne de commande zipfile, 595
MetaPathFinder (*classe dans importlib.abc*), 2068
metavar (*attribut optparse.Option*), 908
MetavarTypeHelpFormatter (*classe dans argparse*), 853
method
    object, 102
method (*attribut urllib.request.Request*), 1432
method_calls (*attribut unittest.mock.Mock*), 1793
methodcaller() (*dans le module operator*), 441
MethodDescriptorType (*dans le module types*), 302
méthode, **2239**
    magic, 2238
    special, 2243
méthode magique, **2238**
méthode spéciale, **2243**
methodHelp() (*méthode xmlrpc.client.ServerProxy.system*), 1523
methods
    bytearray, 67
    bytes, 67
    string, 54
methods (*attribut pyclbr.Class*), 2154
methodSignature() (*méthode xmlrpc.client.ServerProxy.system*), 1523
MethodType (*dans le module types*), 302
MethodWrapperType (*dans le module types*), 302
metrics() (*méthode tkinter.font.Font*), 1633
MFD_ALLOW_SEALING (*dans le module os*), 706
MFD_CLOEXEC (*dans le module os*), 706
MFD_HUGE_1GB (*dans le module os*), 706
MFD_HUGE_1MB (*dans le module os*), 706
MFD_HUGE_2GB (*dans le module os*), 706
MFD_HUGE_2MB (*dans le module os*), 706
MFD_HUGE_8MB (*dans le module os*), 706
MFD_HUGE_16GB (*dans le module os*), 706
MFD_HUGE_16MB (*dans le module os*), 706
MFD_HUGE_32MB (*dans le module os*), 706
MFD_HUGE_64KB (*dans le module os*), 706
MFD_HUGE_256MB (*dans le module os*), 706
MFD_HUGE_512KB (*dans le module os*), 706
MFD_HUGE_512MB (*dans le module os*), 706
MFD_HUGE_MASK (*dans le module os*), 706
MFD_HUGE_SHIFT (*dans le module os*), 706
MFD_HUGETLB (*dans le module os*), 706
MH (*classe dans mailbox*), 1324

MHMessage (*classe dans mailbox*), 1330
microsecond (*attribut datetime.datetime*), 222
microsecond (*attribut datetime.time*), 230
microseconds (*attribut datetime.timedelta*), 211
MIME
    base64 encoding, 1339
    content type, 1336
    headers, 1336
    quoted-printable encoding, 1345
MIMEApplication (*classe dans email.mime.application*), 1297
MIMEAudio (*classe dans email.mime.audio*), 1297
MIMEBase (*classe dans email.mime.base*), 1296
MIMEImage (*classe dans email.mime.image*), 1297
MIMEMessage (*classe dans email.mime.message*), 1298
MIMEMultipart (*classe dans email.mime.multipart*), 1296
MIMENonMultipart (*classe dans email.mime.nonmultipart*), 1296
MIMEPart (*classe dans email.message*), 1258
MIMEText (*classe dans email.mime.text*), 1298
mimetypes
    module, 1336
MimeTypes (*classe dans mimetypes*), 1338
MIMEVersionHeader (*classe dans email.headerregistry*), 1275
min
    built-in function, 47
min (*attribut datetime.date*), 214
min (*attribut datetime.datetime*), 221
min (*attribut datetime.time*), 229
min (*attribut datetime.timedelta*), 210
min (*attribut sys.float_info*), 1938
min()
    built-in function, 22
min() (*méthode decimal.Context*), 368
min() (*méthode decimal.Decimal*), 360
min_10_exp (*attribut sys.float_info*), 1938
MIN_EMIN (*dans le module decimal*), 369
MIN_ETINY (*dans le module decimal*), 369
min_exp (*attribut sys.float_info*), 1938
min_mag() (*méthode decimal.Context*), 368
min_mag() (*méthode decimal.Decimal*), 360
MINEQUAL (*dans le module token*), 2145
MINIMUM_SUPPORTED (*attribut ssl.TLSVersion*), 1201
minimum_version (*attribut ssl.SSLContext*), 1212
minor (*attribut email.headerregistry.MIMEVersionHeader*), 1276
minor() (*dans le module os*), 693
MINUS (*dans le module token*), 2144
minus() (*méthode decimal.Context*), 368
minute (*attribut datetime.datetime*), 222
minute (*attribut datetime.time*), 230
MINYEAR (*dans le module datetime*), 208
mirrored() (*dans le module unicodedata*), 173
misc_header (*attribut cmd.Cmd*), 1610
--missing
    option de ligne de commande trace, 1899
MISSING (*attribut contextvars.Token*), 1054
MISSING (*dans le module dataclasses*), 1984
MISSING (*dans le module sys.monitoring*), 1959
MISSING_C_DOCSTRINGS (*dans le module test.support*), 1848
missing_compiler_executable() (*dans le module test.support*), 1854
MissingSectionHeaderError, 639
mkd() (*méthode ftplib.FTP*), 1469
mkdir() (*dans le module os*), 691
mkdir() (*méthode pathlib.Path*), 466
mkdir() (*méthode zipfile.ZipFile*), 590
mkdtemp() (*dans le module tempfile*), 490
mkfifo() (*dans le module os*), 692
mknod() (*dans le module os*), 693
mkstemp() (*dans le module tempfile*), 489
mktemp() (*dans le module tempfile*), 492
mktime() (*dans le module time*), 746
mktime_tz() (*dans le module email.utils*), 1305
mlsd() (*méthode ftplib.FTP*), 1468
mmap
    module, 1243
mmap (*classe dans mmap*), 1244
MMDF (*classe dans mailbox*), 1326
MMDFMessage (*classe dans mailbox*), 1333
Mock (*classe dans unittest.mock*), 1786
mock_add_spec() (*méthode unittest.mock.Mock*), 1789
mock_calls (*attribut unittest.mock.Mock*), 1793
mock_open() (*dans le module unittest.mock*), 1819
Mod (*classe dans ast*), 2110
mod() (*dans le module operator*), 439
--mode
    option de ligne de commande ast, 2138
mode (*attribut bz2.BZ2File*), 576
mode (*attribut gzip.GzipFile*), 572
mode (*attribut io.FileIO*), 737
mode (*attribut lzma.LZMAFile*), 580
mode (*attribut statistics.NormalDist*), 404
mode (*attribut tarfile.TarInfo*), 603
mode binaire, 25
mode() (*dans le module statistics*), 399
mode() (*dans le module turtle*), 1599
modes
    file, 23
modf() (*dans le module math*), 341
modified() (*méthode urllib.robotparser.RobotFileParser*), 1453
modify() (*méthode select.devpoll*), 1226
modify() (*méthode select.epoll*), 1227
modify() (*méthode selectors.BaseSelector*), 1232
modify() (*méthode select.poll*), 1228
module, **2239**
    __future__, 2020
    __main__, 1966, 2064
    _locale, 1561

_thread, 1056
_tkinter, 1622
abc, 2004
aifc, 2221
annotationlib, 2045
argparse, 849
array, 65, 289
ast, 2101
asynchat, 2221
asyncio, 1061
asyncore, 2221
atexit, 2009
audioop, 2222
base64, 1339, 1343
bdb, 1869, 1876
binascii, 1343
bisect, 286
builtins, 33, 1965
bz2, 574
calendar, 250
cgi, 2222
cgitb, 2222
chunk, 2222
cmath, 347
cmd, 1608, 1876
code, 2053
codecs, 189
codeop, 2055
collections, 258
collections.abc, 276
colorsys, 1552
compileall, 2156
concurrent.futures, 1020
configparser, 620
contextlib, 1989
contextvars, 1052
copy, 306, 525
copyreg, 525
cProfile, 1888
crypt, 2222
csv, 613
ctypes, 812
curses, 923
curses.ascii, 949
curses.panel, 953
curses.textpad, 948
dataclasses, 1978
datetime, 207
dbm, 531
dbm.dumb, 535
dbm.gnu, 527, 533
dbm.ndbm, 527, 534
dbm.sqlite3, 532
decimal, 351
difflib, 155
dis, 2160
distutils, 2222
doctest, 1728
email, 1249
email.charset, 1301
email.contentmanager, 1279
email.encoders, 1303
email.errors, 1272
email.generator, 1262
email.header, 1298
email.headerregistry, 1273
email.iterators, 1306
email.message, 1250
email.mime, 1296
email.mime.application, 1297
email.mime.audio, 1297
email.mime.base, 1296
email.mime.image, 1297
email.mime.message, 1298
email.mime.multipart, 1296
email.mime.nonmultipart, 1296
email.mime.text, 1298
email.parser, 1258
email.policy, 1264
email.utils, 1304
encodings.idna, 204
encodings.mbcs, 205
encodings.utf_8_sig, 205
ensurepip, 1913
enum, 316
errno, 111, 804
faulthandler, 1874
fcntl, 2204
filecmp, 485
fileinput, 921
fnmatch, 495
fractions, 380
ftplib, 1465
functools, 426
gc, 2021
getopt, 2217
getpass, 920
gettext, 1553
glob, 492, 495
graphlib, 331
grp, 2199
gzip, 571
hashlib, 645
heapq, 282
hmac, 657
html, 1347
html.entities, 1352
html.parser, 1347
http, 1454
http.client, 1457
http.cookiejar, 1512
http.cookies, 1509
http.server, 1502
idlelib, 1670
imaplib, 1474
imghdr, 2223

imp, 2223
importlib, 2066
importlib.abc, 2068
importlib.machinery, 2074
importlib.metadata, 2090
importlib.resources, 2085
importlib.resources.abc, 2088
importlib.util, 2079
inspect, 2025
io, 730
ipaddress, 1534
itertools, 409
json, 1307
json.tool, 1316
keyword, 2147
linecache, 496
locale, 1561
logging, 756
logging.config, 774
logging.handlers, 786
lzma, 579
mailbox, 1317
mailcap, 2223
marshal, 529
math, 40, 338, 350
mimetypes, 1336
mmap, 1243
modulefinder, 2062
msilib, 2223
msvcrt, 2183
multiprocessing, 969
multiprocessing.connection, 1001
multiprocessing.dummy, 1004
multiprocessing.managers, 991
multiprocessing.pool, 998
multiprocessing.shared_memory, 1014
multiprocessing.sharedctypes, 989
netrc, 641
nis, 2223
nntplib, 2223
numbers, 335
operator, 437
optparse, 894
os, 663, 2197
os.path, 472
ossaudiodev, 2224
pathlib, 445
pathlib.types, 472
pdb, 1876
pickle, 306, 509, 525, 526, 529
pickletools, 2181
pipes, 2224
pkgutil, 2059
platform, 800
plistlib, 642
poplib, 1471
posix, 2197
pprint, 307
profile, 1888
pstats, 1889
pty, 679, 2202
pwd, 474, 2198
py_compile, 2154
pyclbr, 2152
pydoc, 1724
pyexpat, 1403
queue, 1049
quopri, 1345
random, 383
re, 54, 132, 495
readline, 175
reprlib, 313
resource, 2206
rlcompleter, 180
runpy, 2063
sched, 1047
search path, 496, 1946, 2049
secrets, 659
select, 1224
selectors, 1231
shelve, 526, 529
shlex, 1613
shutil, 497
signal, 1059, 1234
site, 2049
sitecustomize, 2050
smtpd, 2224
smtplib, 1482
sndhdr, 2224
socket, 1162, 1413
socketserver, 1493
spwd, 2224
sqlite3, 536
ssl, 1189
stat, 479, 698
statistics, 393
string, 121
stringprep, 174
struct, 181, 1185
subprocess, 1028
sunau, 2224
symtable, 2139
sys, 25, 1929
sysconfig, 1959
syslog, 2210
sys.monitoring, 1955
tabnanny, 2151
tarfile, 595
telnetlib, 2225
tempfile, 487
termios, 2200
test, 1844
test.regrtest, 1846
test.support, 1847
test.support.bytecode_helper, 1858
test.support.import_helper, 1861

test.support.os_helper, 1859
test.support.script_helper, 1857
test.support.socket_helper, 1856
test.support.threading_helper, 1858
test.support.warnings_helper, 1862
textwrap, 168
threading, 955
time, 743
timeit, 1893
tkinter, 1619
tkinter.colorchooser, 1632
tkinter.commondialog, 1637
tkinter.dnd, 1639
tkinter.filedialog, 1634
tkinter.font, 1633
tkinter.messagebox, 1637
tkinter.scrolledtext, 1639
tkinter.simpledialog, 1634
tkinter.ttk, 1640
token, 2143
tokenize, 2147
tomllib, 639
trace, 1898
traceback, 2010
tracemalloc, 1901
tty, 2201
turtle, 1571
turtledemo, 1606
types, 103, 299
typing, 1671
unicodedata, 172
unittest, 1751
unittest.mock, 1783
urllib, 1426
urllib.error, 1452
urllib.parse, 1443
urllib.request, 1426, 1457
urllib.response, 1442
urllib.robotparser, 1452
usercustomize, 2050
uu, 2225
uuid, 1488
venv, 1915
warnings, 1971
wave, 1549
weakref, 292
webbrowser, 1413
winreg, 2186
winsound, 2194
wsgiref, 1416
wsgiref.handlers, 1421
wsgiref.headers, 1418
wsgiref.simple_server, 1419
wsgiref.types, 1424
wsgiref.util, 1416
wsgiref.validate, 1420
xdrlib, 2225
xml, 1353
xml.dom, 1374
xml.dom.minidom, 1384
xml.dom.pulldom, 1389
xml.etree.ElementInclude, 1366
xml.etree.ElementTree, 1354
xml.parsers.expat, 1403
xml.parsers.expat.errors, 1410
xml.parsers.expat.model, 1409
xmlrpc, 1521
xmlrpc.client, 1521
xmlrpc.server, 1528
xml.sax, 1391
xml.sax.handler, 1392
xml.sax.saxutils, 1398
xml.sax.xmlreader, 1399
zipapp, 1924
zipfile, 584
zipimport, 2057
zlib, 567
zoneinfo, 245
module (*attribut pyclbr.Class*), 2153
module (*attribut pyclbr.Function*), 2153
MODULE (*attribut symtable.SymbolTableType*), 2139
Module (*classe dans ast*), 2106
Module browser, 1659
module d'extension, **2234**
module_from_spec() (*dans le module importlib.util*), 2081
modulefinder
module, 2062
ModuleFinder (*classe dans modulefinder*), 2062
ModuleInfo (*classe dans pkgutil*), 2059
ModuleNotFoundError, 110
modules (*attribut modulefinder.ModuleFinder*), 2062
modules (*dans le module sys*), 1946
modules_cleanup() (*dans le module test.support.import_helper*), 1862
modules_setup() (*dans le module test.support.import_helper*), 1861
ModuleSpec (*classe dans importlib.machinery*), 2078
ModuleType (*classe dans types*), 302
modulus (*attribut sys.hash_info*), 1942
MON_1 (*dans le module locale*), 1563
MON_2 (*dans le module locale*), 1563
MON_3 (*dans le module locale*), 1563
MON_4 (*dans le module locale*), 1563
MON_5 (*dans le module locale*), 1563
MON_6 (*dans le module locale*), 1563
MON_7 (*dans le module locale*), 1563
MON_8 (*dans le module locale*), 1563
MON_9 (*dans le module locale*), 1563
MON_10 (*dans le module locale*), 1563
MON_11 (*dans le module locale*), 1563
MON_12 (*dans le module locale*), 1563
MONDAY (*dans le module calendar*), 255
monotonic() (*dans le module time*), 746
monotonic_ns() (*dans le module time*), 747
month

option de ligne de commande calendar, 258
month (*attribut calendar.IllegalMonthError*), 256
month (*attribut datetime.date*), 214
month (*attribut datetime.datetime*), 222
Month (*classe dans calendar*), 256
month() (*dans le module calendar*), 254
month_abbr (*dans le module calendar*), 255
month_name (*dans le module calendar*), 255
monthcalendar() (*dans le module calendar*), 254
monthdatescalendar() (*méthode calendar.Calendar*), 251
monthdays2calendar() (*méthode calendar.Calendar*), 251
monthdayscalendar() (*méthode calendar.Calendar*), 251
monthrange() (*dans le module calendar*), 254
--months
option de ligne de commande calendar, 258
Morsel (*classe dans http.cookies*), 1510
most_common() (*méthode collections.Counter*), 262
mouseinterval() (*dans le module curses*), 927
mousemask() (*dans le module curses*), 927
move() (*dans le module shutil*), 501
move() (*méthode curses.panel.Panel*), 954
move() (*méthode curses.window*), 935
move() (*méthode mmap.mmap*), 1246
move() (*méthode pathlib.Path*), 468
move() (*méthode tkinter.ttk.Treeview*), 1653
move_into() (*méthode pathlib.Path*), 468
move_to_end() (*méthode collections.OrderedDict*), 273
MozillaCookieJar (*classe dans http.cookiejar*), 1516
MRO, **2239**
msg (*attribut http.client.HTTPResponse*), 1463
msg (*attribut json.JSONDecodeError*), 1314
msg (*attribut netrc.NetrcParseError*), 641
msg (*attribut re.PatternError*), 146
msg (*attribut tomllib.TOMLDecodeError*), 640
msg (*attribut traceback.TracebackException*), 2014
msilib
module, 2223
msvcrt
module, 2183
mtime (*attribut gzip.GzipFile*), 572
mtime (*attribut tarfile.TarInfo*), 603
mtime() (*méthode urllib.robotparser.RobotFileParser*), 1453
mul() (*dans le module operator*), 439
Mult (*classe dans ast*), 2110
MultiCall (*classe dans xmlrpc.client*), 1526
MULTILINE (*dans le module re*), 141
MultilineContinuationError, 639
multimode() (*dans le module statistics*), 399
MultipartConversionError, 1272
multiply() (*méthode decimal.Context*), 368
multiprocessing
module, 969
multiprocessing.connection
module, 1001
multiprocessing.dummy
module, 1004
multiprocessing.Manager()
built-in function, 991
multiprocessing.managers
module, 991
multiprocessing.pool
module, 998
multiprocessing.shared_memory
module, 1014
multiprocessing.sharedctypes
module, 989
mutable, **2239**
sequence types, 49
mutable sequence
loop over, 47
MutableMapping (*classe dans collections.abc*), 279
MutableMapping (*classe dans typing*), 1721
MutableSequence (*classe dans collections.abc*), 279
MutableSequence (*classe dans typing*), 1721
MutableSet (*classe dans collections.abc*), 279
MutableSet (*classe dans typing*), 1721
mvderwin() (*méthode curses.window*), 935
mvwin() (*méthode curses.window*), 935
myrights() (*méthode imaplib.IMAP4*), 1478

N

-N
option de ligne de commande uuid, 1492
-n
option de ligne de commande timeit, 1896
option de ligne de commande uuid, 1492
n-uplet nommé, **2239**
N_TOKENS (*dans le module token*), 2146
n_waiting (*attribut asyncio.Barrier*), 1097
n_waiting (*attribut threading.Barrier*), 968
NAK (*dans le module curses.ascii*), 951
--name
option de ligne de commande uuid, 1492
name (*attribut bz2.BZ2File*), 576
name (*attribut codecs.CodecInfo*), 189
name (*attribut contextvars.ContextVar*), 1053
name (*attribut doctest.DocTest*), 1743
name (*attribut email.headerregistry.BaseHeader*), 1273
name (*attribut enum.Enum*), 319
name (*attribut gzip.GzipFile*), 572
name (*attribut hashlib.hash*), 647
name (*attribut hmac.HMAC*), 658
name (*attribut http.cookiejar.Cookie*), 1519
name (*attribut ImportError*), 109
name (*attribut importlib.abc.FileLoader*), 2071
name (*attribut importlib.abc.Traversable*), 2073
name (*attribut importlib.machinery.AppleFrameworkLoader*),

2079
name (*attribut importlib.machinery.ExtensionFileLoader*), 2077
name (*attribut importlib.machinery.ModuleSpec*), 2078
name (*attribut importlib.machinery.SourceFileLoader*), 2076
name (*attribut importlib.machinery.SourcelessFileLoader*), 2077
name (*attribut importlib.resources.abc.Traversable*), 2089
name (*attribut inspect.Parameter*), 2034
name (*attribut io.FileIO*), 737
name (*attribut logging.Logger*), 757
name (*attribut lzma.LZMAFile*), 580
name (*attribut multiprocessing.Process*), 977
name (*attribut multiprocessing.shared_memory.SharedMemory*), 1015
name (*attribut os.DirEntry*), 696
name (*attribut pathlib.PurePath*), 452
name (*attribut pyclbr.Class*), 2153
name (*attribut pyclbr.Function*), 2153
name (*attribut sys.thread_info*), 1953
name (*attribut tarfile.TarInfo*), 603
name (*attribut tempfile.TemporaryDirectory*), 489
name (*attribut threading.Thread*), 960
name (*attribut traceback.FrameSummary*), 2016
name (*attribut webbrowser.controller*), 1415
name (*attribut xml.dom.Attr*), 1381
name (*attribut xml.dom.DocumentType*), 1379
name (*attribut zipfile.Path*), 590
Name (*classe dans ast*), 2109
name (*dans le module os*), 664
NAME (*dans le module token*), 2143
name() (*dans le module unicodedata*), 172
name2codepoint (*dans le module html.entities*), 1352
Named Shared Memory, 1014
NAMED_FLAGS (*attribut enum.EnumCheck*), 326
NamedExpr (*classe dans ast*), 2112
NamedTemporaryFile() (*dans le module tempfile*), 487
NamedTuple (*classe dans typing*), 1703
namedtuple() (*dans le module collections*), 269
NameError, 110
namelist() (*méthode zipfile.ZipFile*), 587
nameprep() (*dans le module encodings.idna*), 205
namer (*attribut logging.handlers.BaseRotatingHandler*), 788
namereplace
    error handler's name, 192
namereplace_errors() (*dans le module codecs*), 194
names() (*dans le module tkinter.font*), 1634
--namespace
    option de ligne de commande uuid, 1492
Namespace (*classe dans argparse*), 871
Namespace (*classe dans multiprocessing.managers*), 994
namespace() (*méthode imaplib.IMAP4*), 1478
Namespace() (*méthode multiprocessing.managers.SyncManager*), 993
NAMESPACE_DNS (*dans le module uuid*), 1491
NAMESPACE_OID (*dans le module uuid*), 1491
NAMESPACE_URL (*dans le module uuid*), 1491
NAMESPACE_X500 (*dans le module uuid*), 1491
NamespaceErr, 1382
NamespaceLoader (*classe dans importlib.machinery*), 2078
namespaceURI (*attribut xml.dom.Node*), 1377
nametofont() (*dans le module tkinter.font*), 1634
NaN, 16
nan (*attribut sys.hash_info*), 1943
nan (*dans le module cmath*), 350
nan (*dans le module math*), 346
nanj (*dans le module cmath*), 350
NannyNag, 2152
napms() (*dans le module curses*), 927
nargs (*attribut optparse.Option*), 908
native_id (*attribut threading.Thread*), 960
nbytes (*attribut memoryview*), 86
ncurses_version (*dans le module curses*), 937
ND (*attribut inspect.BufferFlags*), 2044
ndiff() (*dans le module difflib*), 158
ndim (*attribut memoryview*), 87
ne() (*dans le module operator*), 437
needs_input (*attribut bz2.BZ2Decompressor*), 577
needs_input (*attribut lzma.LZMADecompressor*), 582
neg() (*dans le module operator*), 439
netmask (*attribut ipaddress.IPv4Network*), 1541
netmask (*attribut ipaddress.IPv6Network*), 1544
NetmaskValueError, 1547
netrc
    module, 641
netrc (*classe dans netrc*), 641
NetrcParseError, 641
netscape (*attribut http.cookiejar.CookiePolicy*), 1517
network (*attribut ipaddress.IPv4Interface*), 1545
network (*attribut ipaddress.IPv6Interface*), 1546
network_address (*attribut ipaddress.IPv4Network*), 1541
network_address (*attribut ipaddress.IPv6Network*), 1544
Never (*dans le module typing*), 1684
NEVER_EQ (*dans le module test.support*), 1849
new() (*dans le module hashlib*), 646
new() (*dans le module hmac*), 657
new_child() (*méthode collections.ChainMap*), 259
new_class() (*dans le module types*), 300
new_event_loop() (*dans le module asyncio*), 1108
new_event_loop() (*méthode asyncio.AbstractEventLoopPolicy*), 1149
new_panel() (*dans le module curses.panel*), 953
NEWLINE (*dans le module token*), 2144
newlines (*attribut io.TextIOBase*), 740
newpad() (*dans le module curses*), 927
NewType (*classe dans typing*), 1704
newwin() (*dans le module curses*), 927
next (*pdb command*), 1881
next()
    built-in function, 22

next() (*méthode tarfile.TarFile*), 600
next() (*méthode tkinter.ttk.Treeview*), 1653
next_minus() (*méthode decimal.Context*), 368
next_minus() (*méthode decimal.Decimal*), 360
next_plus() (*méthode decimal.Context*), 368
next_plus() (*méthode decimal.Decimal*), 360
next_toward() (*méthode decimal.Context*), 368
next_toward() (*méthode decimal.Decimal*), 360
nextafter() (*dans le module math*), 342
nextfile() (*dans le module fileinput*), 922
nextkey() (*méthode dbm.gnu.gdbm*), 533
nextSibling (*attribut xml.dom.Node*), 1377
ngettext() (*dans le module gettext*), 1554
ngettext() (*méthode gettext.GNUTranslations*), 1557
ngettext() (*méthode gettext.NullTranslations*), 1556
nice() (*dans le module os*), 716
NIL (*dans le module uuid*), 1491
nis
    module, 2223
NL (*dans le module curses.ascii*), 950
NL (*dans le module token*), 2146
nl() (*dans le module curses*), 927
nl_langinfo() (*dans le module locale*), 1563
nlargest() (*dans le module heapq*), 283
nlst() (*méthode ftplib.FTP*), 1468
nntplib
    module, 2223
NO (*dans le module tkinter.messagebox*), 1638
no_animation() (*dans le module turtle*), 1597
no_cache() (*méthode de la classe zoneinfo.ZoneInfo*), 247
NO_EVENTS (*monitoring event*), 1957
no_proxy, 1430
no_site (*attribut sys.flags*), 1936
no_tracing() (*dans le module test.support*), 1853
no_type_check() (*dans le module typing*), 1714
no_type_check_decorator() (*dans le module typing*), 1714
no_user_site (*attribut sys.flags*), 1936
nocbreak() (*dans le module curses*), 928
NoDataAllowedErr, 1383
node (*attribut uuid.UUID*), 1490
node() (*dans le module platform*), 800
NoDefault (*dans le module typing*), 1718
nodelay() (*méthode curses.window*), 935
nodeName (*attribut xml.dom.Node*), 1377
NodeTransformer (*classe dans ast*), 2136
nodeType (*attribut xml.dom.Node*), 1376
nodeValue (*attribut xml.dom.Node*), 1377
NodeVisitor (*classe dans ast*), 2136
noecho() (*dans le module curses*), 928
--no-ensure-ascii
    option de ligne de commande json, 1317
NOEXPR (*dans le module locale*), 1564
NOFLAG (*dans le module re*), 141
--no-indent
    option de ligne de commande json, 1317
nom qualifié, **2242**
nombre complexe, **2232**
nombre de références, **2242**
NoModificationAllowedErr, 1383
NonCallableMagicMock (*classe dans unittest.mock*), 1814
NonCallableMock (*classe dans unittest.mock*), 1794
None (*Built-in object*), 37
None (*variable de base*), 35
NoneType (*dans le module types*), 301
nonl() (*dans le module curses*), 928
Nonlocal (*classe dans ast*), 2132
nonmember() (*dans le module enum*), 330
noop() (*méthode imaplib.IMAP4*), 1478
noop() (*méthode poplib.POP3*), 1473
NoOptionError, 638
NOP (*opcode*), 2166
noqiflush() (*dans le module curses*), 928
noraw() (*dans le module curses*), 928
--no-report
    option de ligne de commande trace, 1899
NoReturn (*dans le module typing*), 1684
NORMAL (*dans le module tkinter.font*), 1633
NORMAL_PRIORITY_CLASS (*dans le module subprocess*), 1041
NormalDist (*classe dans statistics*), 404
normalize() (*dans le module locale*), 1566
normalize() (*dans le module unicodedata*), 173
normalize() (*méthode decimal.Context*), 368
normalize() (*méthode decimal.Decimal*), 360
normalize() (*méthode xml.dom.Node*), 1378
NORMALIZE_WHITESPACE (*dans le module doctest*), 1736
normalvariate() (*dans le module random*), 387
normcase() (*dans le module os.path*), 476
normpath() (*dans le module os.path*), 476
NoSectionError, 638
NoSuchMailboxError, 1334
not
    operator, 38
Not (*classe dans ast*), 2110
not in
    operator, 38, 47
not_() (*dans le module operator*), 438
NotADirectoryError, 115
notationDecl() (*méthode xml.sax.handler.DTDHandler*), 1397
NotationDeclHandler() (*méthode xml.parsers.expat.xmlparser*), 1407
notations (*attribut xml.dom.DocumentType*), 1379
NotConnected, 1458
Notebook (*classe dans tkinter.ttk*), 1647
NotEmptyError, 1334
NotEq (*classe dans ast*), 2111
NOTEQUAL (*dans le module token*), 2145
NotFoundErr, 1383
notify() (*méthode asyncio.Condition*), 1094
notify() (*méthode threading.Condition*), 965

notify_all() (*méthode asyncio.Condition*), 1095
notify_all() (*méthode threading.Condition*), 965
notimeout() (*méthode curses.window*), 935
NotImplemented (*variable de base*), 35
NotImplementedError, 110
NotImplementedType (*dans le module types*), 302
NotIn (*classe dans ast*), 2111
NotRequired (*dans le module typing*), 1689
NOTSET (*dans le module logging*), 762
NotStandaloneHandler() (*méthode xml.parsers.expat.xmlparser*), 1408
NotSupportedErr, 1383
NotSupportedError, 557
--no-type-comments
    option de ligne de commande ast, 2139
noutrefresh() (*méthode curses.window*), 935
nouvelle classe, **2240**
NOVEMBER (*dans le module calendar*), 256
now() (*méthode de la classe datetime.datetime*), 218
npgettext() (*dans le module gettext*), 1554
npgettext() (*méthode gettext.GNUTranslations*), 1557
npgettext() (*méthode gettext.NullTranslations*), 1556
NSIG (*dans le module signal*), 1237
nsmallest() (*dans le module heapq*), 283
NT_OFFSET (*dans le module token*), 2146
NTEventLogHandler (*classe dans logging.handlers*), 795
ntohl() (*dans le module socket*), 1175
ntohs() (*dans le module socket*), 1175
ntransfercmd() (*méthode ftplib.FTP*), 1468
NUL (*dans le module curses.ascii*), 950
nullcontext() (*dans le module contextlib*), 1992
NullHandler (*classe dans logging*), 787
NullTranslations (*classe dans gettext*), 1555
num_addresses (*attribut ipaddress.IPv4Network*), 1541
num_addresses (*attribut ipaddress.IPv6Network*), 1544
num_tickets (*attribut ssl.SSLContext*), 1212
--number
    option de ligne de commande timeit, 1896
Number (*classe dans numbers*), 335
NUMBER (*dans le module token*), 2144
number_class() (*méthode decimal.Context*), 368
number_class() (*méthode decimal.Decimal*), 361
numbers
    module, 335
numerator (*attribut fractions.Fraction*), 381
numerator (*attribut numbers.Rational*), 336
numeric
    conversions, 40
    literals, 39
    object, 38
    types, operations on, 39
numeric() (*dans le module unicodedata*), 172
numinput() (*dans le module turtle*), 1599

O

-O
    option de ligne de commande dis, 2161
-o
    option de ligne de commande compileall, 2157
    option de ligne de commande pickletools, 2182
    option de ligne de commande zipapp, 1925
O_APPEND (*dans le module os*), 678
O_ASYNC (*dans le module os*), 678
O_BINARY (*dans le module os*), 678
O_CLOEXEC (*dans le module os*), 678
O_CREAT (*dans le module os*), 678
O_DIRECT (*dans le module os*), 678
O_DIRECTORY (*dans le module os*), 678
O_DSYNC (*dans le module os*), 678
O_EVTONLY (*dans le module os*), 678
O_EXCL (*dans le module os*), 678
O_EXLOCK (*dans le module os*), 678
O_FSYNC (*dans le module os*), 678
O_NDELAY (*dans le module os*), 678
O_NOATIME (*dans le module os*), 678
O_NOCTTY (*dans le module os*), 678
O_NOFOLLOW (*dans le module os*), 678
O_NOFOLLOW_ANY (*dans le module os*), 678
O_NOINHERIT (*dans le module os*), 678
O_NONBLOCK (*dans le module os*), 678
O_PATH (*dans le module os*), 678
O_RANDOM (*dans le module os*), 678
O_RDONLY (*dans le module os*), 678
O_RDWR (*dans le module os*), 678
O_RSYNC (*dans le module os*), 678
O_SEQUENTIAL (*dans le module os*), 678
O_SHLOCK (*dans le module os*), 678
O_SHORT_LIVED (*dans le module os*), 678
O_SYMLINK (*dans le module os*), 678
O_SYNC (*dans le module os*), 678
O_TEMPORARY (*dans le module os*), 678
O_TEXT (*dans le module os*), 678
O_TMPFILE (*dans le module os*), 678
O_TRUNC (*dans le module os*), 678
O_WRONLY (*dans le module os*), 678
obj (*attribut memoryview*), 86
object
    Boolean, 38
    bytearray, 49, 65, 66
    bytes, 65
    code, 103, 529
    complex number, 38
    dictionary, 90
    floating-point, 38
    GenericAlias, 96
    integer, 38
    io.StringIO, 53
    list, 49
    mapping, 90

memoryview, 65
method, 102
numeric, 38
range, 51
sequence, 47
set, 88
socket, 1162
string, 52
traceback, 1934, 2010
tuple, 48, 50
type, 32
Union, 100
object (*attribut UnicodeError*), 114
object (*classe de base*), 23
objects
comparing, 38
flattening, 509
marshalling, 509
persistent, 509
pickling, 509
serializing, 509
objet, **2240**
objet fichier, **2234**
objet fichier-compatible, **2234**
objet octet-compatible, **2231**
objet simili-chemin, **2241**
oct()
built-in function, 23
octal
literals, 39
octdigits (*dans le module string*), 122
OCTOBER (*dans le module calendar*), 256
offset (*attribut SyntaxError*), 112
offset (*attribut tarfile.TarInfo*), 604
offset (*attribut traceback.TracebackException*), 2014
offset (*attribut xml.parsers.expat.ExpatError*), 1408
offset_data (*attribut tarfile.TarInfo*), 604
OK (*dans le module curses*), 937
OK (*dans le module tkinter.messagebox*), 1638
ok_command() (*méthode tkinter.filedialog.LoadFileDialog*), 1636
ok_command() (*méthode tkinter.filedialog.SaveFileDialog*), 1636
ok_event() (*méthode tkinter.filedialog.FileDialog*), 1636
OKCANCEL (*dans le module tkinter.messagebox*), 1639
old_value (*attribut contextvars.Token*), 1054
OleDLL (*classe dans ctypes*), 833
on_motion() (*méthode tkinter.dnd.DndHandler*), 1640
on_release() (*méthode tkinter.dnd.DndHandler*), 1640
onclick() (*dans le module turtle*), 1598
ondrag() (*dans le module turtle*), 1593
onecmd() (*méthode cmd.Cmd*), 1609
onkey() (*dans le module turtle*), 1597
onkeypress() (*dans le module turtle*), 1598
onkeyrelease() (*dans le module turtle*), 1597
onrelease() (*dans le module turtle*), 1592
onscreenclick() (*dans le module turtle*), 1598
ontimer() (*dans le module turtle*), 1598
OP (*dans le module token*), 2146
OP_ALL (*dans le module ssl*), 1197
OP_CIPHER_SERVER_PREFERENCE (*dans le module ssl*), 1198
OP_ENABLE_KTLS (*dans le module ssl*), 1199
OP_ENABLE_MIDDLEBOX_COMPAT (*dans le module ssl*), 1198
OP_IGNORE_UNEXPECTED_EOF (*dans le module ssl*), 1199
OP_LEGACY_SERVER_CONNECT (*dans le module ssl*), 1199
OP_NO_COMPRESSION (*dans le module ssl*), 1199
OP_NO_RENEGOTIATION (*dans le module ssl*), 1198
OP_NO_SSLv2 (*dans le module ssl*), 1197
OP_NO_SSLv3 (*dans le module ssl*), 1197
OP_NO_TICKET (*dans le module ssl*), 1199
OP_NO_TLSv1 (*dans le module ssl*), 1198
OP_NO_TLSv1_1 (*dans le module ssl*), 1198
OP_NO_TLSv1_2 (*dans le module ssl*), 1198
OP_NO_TLSv1_3 (*dans le module ssl*), 1198
OP_SINGLE_DH_USE (*dans le module ssl*), 1198
OP_SINGLE_ECDH_USE (*dans le module ssl*), 1198
Open (*classe dans tkinter.filedialog*), 1635
open()
built-in function, 23
open() (*dans le module bz2*), 574
open() (*dans le module codecs*), 190
open() (*dans le module dbm*), 531
open() (*dans le module dbm.dumb*), 535
open() (*dans le module dbm.gnu*), 533
open() (*dans le module dbm.ndbm*), 534
open() (*dans le module dbm.sqlite3*), 532
open() (*dans le module gzip*), 571
open() (*dans le module io*), 732
open() (*dans le module lzma*), 579
open() (*dans le module os*), 677
open() (*dans le module shelve*), 526
open() (*dans le module tarfile*), 596
open() (*dans le module tokenize*), 2149
open() (*dans le module wave*), 1549
open() (*dans le module webbrowser*), 1414
open() (*méthode de la classe tarfile.TarFile*), 600
open() (*méthode imaplib.IMAP4*), 1478
open() (*méthode importlib.abc.Traversable*), 2074
open() (*méthode importlib.resources.abc.Traversable*), 2089
open() (*méthode pathlib.Path*), 462
open() (*méthode urllib.request.OpenerDirector*), 1433
open() (*méthode webbrowser.controller*), 1415
open() (*méthode zipfile.Path*), 590
open() (*méthode zipfile.ZipFile*), 587
open_binary() (*dans le module importlib.resources*), 2086
open_code() (*dans le module io*), 732
open_connection() (*dans le module asyncio*), 1085
open_flags (*dans le module dbm.gnu*), 533

open_new() (*dans le module webbrowser*), 1414
open_new() (*méthode webbrowser.controller*), 1416
open_new_tab() (*dans le module webbrowser*), 1414
open_new_tab() (*méthode webbrowser.controller*), 1416
open_osfhandle() (*dans le module msvcrt*), 2184
open_resource() (*méthode importlib.abc.ResourceReader*), 2073
open_resource() (*méthode importlib.resources.abc.ResourceReader*), 2088
open_text() (*dans le module importlib.resources*), 2086
open_unix_connection() (*dans le module asyncio*), 1086
open_urlresource() (*dans le module test.support*), 1853
OpenerDirector (*classe dans urllib.request*), 1429
OpenKey() (*dans le module winreg*), 2189
OpenKeyEx() (*dans le module winreg*), 2189
openlog() (*dans le module syslog*), 2211
openpty() (*dans le module os*), 679
openpty() (*dans le module pty*), 2202
OpenSSL
    (use in module hashlib), 645
    (use in module ssl), 1189
OPENSSL_VERSION (*dans le module ssl*), 1200
OPENSSL_VERSION_INFO (*dans le module ssl*), 1200
OPENSSL_VERSION_NUMBER (*dans le module ssl*), 1201
operation
    concatenation, 47
    repetition, 47
    slice, 47
    subscript, 47
OperationalError, 556
operations
    bitwise, 40
    Boolean, 37
    masking, 40
    shifting, 40
operations on
    dictionary type, 90
    integer types, 40
    list type, 49
    mapping types, 90
    numeric types, 39
    sequence types, 47, 49
operator
    - (*minus*), 39
    % (*percent*), 39
    & (*ampersand*), 40
    * (*asterisk*), 39
    **, 39
    + (*plus*), 39
    / (*slash*), 39
    //, 39
    < (*less*), 38
    <<, 40
    <=, 38
    !=, 38
    ==, 38
    > (*greater*), 38
    >=, 38
    >>, 40
    ^ (*caret*), 40
    | (*vertical bar*), 40
    ~ (*tilde*), 40
    and, 37, 38
    comparison, 38
    in, 38, 47
    is, 38
    is not, 38
    module, 437
    not, 38
    not in, 38, 47
    or, 37, 38
opmap (*dans le module dis*), 2180
opname (*dans le module dis*), 2180
optim_args_from_interpreter_flags() (*dans le module test.support*), 1850
optimize (*attribut sys.flags*), 1936
optimize() (*dans le module pickletools*), 2182
optimized scope, **2240**
OPTIMIZED_BYTECODE_SUFFIXES (*dans le module importlib.machinery*), 2074
Option (*classe dans optparse*), 907
option de ligne de commande ast
    -a, 2139
    -h, 2138
    --help, 2138
    -i, 2139
    --include-attributes, 2139
    --indent, 2139
    -m, 2138
    --mode, 2138
    --no-type-comments, 2139
option de ligne de commande calendar
    -c, 258
    --css, 258
    -e, 257
    --encoding, 257
    -f, 258
    --first-weekday, 258
    -h, 257
    --help, 257
    -L, 257
    -l, 258
    --lines, 258
    --locale, 257
    -m, 258
    month, 258
    --months, 258
    -s, 258
    --spacing, 258
    -t, 257
    --type, 257
    -w, 258

--width, 258
year, 258
option de ligne de commande compileall
-b, 2156
-d, 2156
directory, 2156
-e, 2157
-f, 2156
file, 2156
--hardlink-dupes, 2157
-i, 2156
--invalidation-mode, 2157
-j, 2157
-l, 2156
-o, 2157
-p, 2156
-q, 2156
-r, 2156
-s, 2156
-x, 2156
option de ligne de commande dis
-C, 2160
-h, 2160
--help, 2160
-O, 2161
-P, 2161
-S, 2161
--show-caches, 2160
--show-offsets, 2161
--show-positions, 2161
--specialized, 2161
option de ligne de commande gzip
--best, 574
-d, 574
--decompress, 574
--fast, 574
file, 574
-h, 574
--help, 574
option de ligne de commande inspect
--details, 2044
option de ligne de commande json
--compact, 1317
-h, 1317
--help, 1317
--indent, 1317
infile, 1316
--json-lines, 1317
--no-ensure-ascii, 1317
--no-indent, 1317
outfile, 1316
--sort-keys, 1316
--tab, 1317
option de ligne de commande pickletools
-a, 2182
--annotate, 2182
--indentlevel, 2182
-l, 2182
-m, 2182
--memo, 2182
-o, 2182
--output, 2182
-p, 2182
--preamble, 2182
option de ligne de commande python--m-py_compile
-, 2155
<file>, 2155
-q, 2155
--quiet, 2155
option de ligne de commande python--m-sqlite3-[-h]-[-v]-[filename]-[sql]
-h, 558
--help, 558
-v, 558
--version, 558
option de ligne de commande random
-c, 392
--choice, 392
-f, 392
--float, 392
-h, 392
--help, 392
-i, 392
--integer, 392
option de ligne de commande site
--user-base, 2052
--user-site, 2052
option de ligne de commande tarfile
-c, 609
--create, 609
-e, 609
--extract, 609
--filter, 609
-l, 609
--list, 609
-t, 609
--test, 609
-v, 609
--verbose, 609
option de ligne de commande timeit
-h, 1896
--help, 1896
-n, 1896
--number, 1896
-p, 1896
--process, 1896
-r, 1896
--repeat, 1896
-s, 1896
--setup, 1896
-u, 1896
--unit, 1896
-v, 1896
--verbose, 1896
option de ligne de commande tokenize

-e, 2149
--exact, 2149
-h, 2149
--help, 2149
option de ligne de commande trace
-C, 1899
-c, 1899
--count, 1899
--coverdir, 1899
-f, 1899
--file, 1899
-g, 1900
--help, 1899
--ignore-dir, 1900
--ignore-module, 1900
-l, 1899
--listfuncs, 1899
-m, 1899
--missing, 1899
--no-report, 1899
-R, 1899
-r, 1899
--report, 1899
-s, 1899
--summary, 1899
-T, 1899
-t, 1899
--timing, 1900
--trace, 1899
--trackcalls, 1899
--version, 1899
option de ligne de commande unittest
-b, 1754
--buffer, 1754
-c, 1754
--catch, 1754
--durations, 1754
-f, 1754
--failfast, 1754
-k, 1754
--locals, 1754
option de ligne de commande unittest-discover
-p, 1755
--pattern, 1755
-s, 1755
--start-directory, 1755
-t, 1755
--top-level-directory, 1755
-v, 1755
--verbose, 1755
option de ligne de commande uuid
-h, 1492
--help, 1492
-N, 1492
-n, 1492
--name, 1492
--namespace, 1492
-u, 1492
--uuid, 1492
option de ligne de commande zipapp
-c, 1925
--compress, 1925
-h, 1925
--help, 1925
--info, 1925
-m, 1925
--main, 1925
-o, 1925
--output, 1925
-p, 1925
--python, 1925
option de ligne de commande zipfile
-c, 595
--create, 595
-e, 595
--extract, 595
-l, 594
--list, 594
--metadata-encoding, 595
-t, 595
--test, 595
Optional (*dans le module typing*), 1687
OptionConflictError, 920
OptionError, 920
OptionGroup (*classe dans optparse*), 902
OptionParser (*classe dans optparse*), 905
options (*attribut doctest.Example*), 1744
options (*attribut ssl.SSLContext*), 1212
Options (*classe dans ssl*), 1199
options() (*méthode configparser.ConfigParser*), 635
OptionValueError, 920
optionxform() (*méthode configparser.ConfigParser*), 637
optparse
module, 894
or
operator, 37, 38
Or (*classe dans ast*), 2111
or_() (*dans le module operator*), 439
ord()
built-in function, 26
ordered_attributes (*attribut xml.parsers.expat.xmlparser*), 1405
OrderedDict (*classe dans collections*), 273
OrderedDict (*classe dans typing*), 1719
ordre de résolution des méthodes, **2239**
orig_argv (*dans le module sys*), 1946
origin (*attribut importlib.machinery.ModuleSpec*), 2078
origin_req_host (*attribut urllib.request.Request*), 1431
origin_server (*attribut wsgiref.handlers.BaseHandler*), 1424
os
module, 663, 2197

os_environ (*attribut wsgiref.handlers.BaseHandler*), 1422
OSError, 111
os.path
    module, 472
ossaudiodev
    module, 2224
OUT_TO_DEFAULT (*dans le module msvcrt*), 2185
OUT_TO_MSGBOX (*dans le module msvcrt*), 2185
OUT_TO_STDERR (*dans le module msvcrt*), 2185
outfile
    option de ligne de commande json, 1316
--output
    option de ligne de commande pickletools, 2182
    option de ligne de commande zipapp, 1925
output (*attribut subprocess.CalledProcessError*), 1031
output (*attribut subprocess.TimeoutExpired*), 1030
output (*attribut unittest.TestCase*), 1766
output() (*méthode http.cookies.BaseCookie*), 1510
output() (*méthode http.cookies.Morsel*), 1511
output_charset (*attribut email.charset.Charset*), 1302
output_codec (*attribut email.charset.Charset*), 1302
output_difference() (*méthode doctest.OutputChecker*), 1747
OutputChecker (*classe dans doctest*), 1747
OutputString() (*méthode http.cookies.Morsel*), 1511
OutsideDestinationError, 598
Overflow (*classe dans decimal*), 371
OverflowError, 111
overlap() (*méthode statistics.NormalDist*), 405
overlaps() (*méthode ipaddress.IPv4Network*), 1541
overlaps() (*méthode ipaddress.IPv6Network*), 1544
overlay() (*méthode curses.window*), 935
overload() (*dans le module typing*), 1713
override() (*dans le module typing*), 1714
overwrite() (*méthode curses.window*), 935
owner() (*méthode pathlib.Path*), 469

P

-P
    option de ligne de commande dis, 2161
-p
    option de ligne de commande compileall, 2156
    option de ligne de commande pickletools, 2182
    option de ligne de commande timeit, 1896
    option de ligne de commande unittest-discover, 1755
    option de ligne de commande zipapp, 1925
p (*pdb command*), 1882
P_ALL (*dans le module os*), 723
P_DETACH (*dans le module os*), 720
P_NOWAIT (*dans le module os*), 720
P_NOWAITO (*dans le module os*), 720
P_OVERLAY (*dans le module os*), 720
P_PGID (*dans le module os*), 723
P_PID (*dans le module os*), 723
P_PIDFD (*dans le module os*), 723
P_WAIT (*dans le module os*), 720
pack() (*dans le module struct*), 182
pack() (*méthode mailbox.MH*), 1324
pack() (*méthode struct.Struct*), 188
pack_into() (*dans le module struct*), 182
pack_into() (*méthode struct.Struct*), 188
package, 2050
PackageMetadata (*classe dans importlib.metadata*), 2092
PackageNotFoundError, 2091
PackagePath (*classe dans importlib.metadata*), 2093
packages_distributions() (*dans le module importlib.metadata*), 2094
packed (*attribut ipaddress.IPv4Address*), 1536
packed (*attribut ipaddress.IPv6Address*), 1538
packing
    binary data, 181
packing (*widgets*), 1627
PAGER, 1725
pair_content() (*dans le module curses*), 928
pair_number() (*dans le module curses*), 928
pairwise() (*dans le module itertools*), 416
paquet, **2240**
paquet classique, **2242**
paquet provisoire, **2241**
paquet-espace de nommage, **2239**
Parameter (*classe dans inspect*), 2034
ParameterizedMIMEHeader (*classe dans email.headerregistry*), 1276
parameters (*attribut inspect.Signature*), 2033
paramètre, **2240**
params (*attribut email.headerregistry.ParameterizedMIMEHeader*), 1276
ParamSpec (*classe dans ast*), 2129
ParamSpec (*classe dans typing*), 1700
ParamSpecArgs (*dans le module typing*), 1701
ParamSpecKwargs (*dans le module typing*), 1701
paramstyle (*dans le module sqlite3*), 541
pardir (*dans le module os*), 728
parent (*attribut importlib.machinery.ModuleSpec*), 2078
parent (*attribut logging.Logger*), 757
parent (*attribut pathlib.PurePath*), 451
parent (*attribut pyclbr.Class*), 2153
parent (*attribut pyclbr.Function*), 2153
parent (*attribut urllib.request.BaseHandler*), 1434
parent() (*méthode tkinter.ttk.Treeview*), 1653
parent_process() (*dans le module multiprocessing*), 983
parentNode (*attribut xml.dom.Node*), 1376
parents (*attribut collections.ChainMap*), 259
parents (*attribut pathlib.PurePath*), 451
paretovariate() (*dans le module random*), 388

parse() (*dans le module ast*), 2134
parse() (*dans le module xml.dom.minidom*), 1385
parse() (*dans le module xml.dom.pulldom*), 1390
parse() (*dans le module xml.etree.ElementTree*), 1363
parse() (*dans le module xml.sax*), 1391
parse() (*méthode doctest.DocTestParser*), 1745
parse() (*méthode email.parser.BytesParser*), 1260
parse() (*méthode email.parser.Parser*), 1260
parse() (*méthode string.Formatter*), 122
parse() (*méthode urllib.robotparser.RobotFileParser*), 1453
parse() (*méthode xml.etree.ElementTree.ElementTree*), 1369
Parse() (*méthode xml.parsers.expat.xmlparser*), 1404
parse() (*méthode xml.sax.xmlreader.XMLReader*), 1400
parse_and_bind() (*dans le module readline*), 176
parse_args() (*méthode argparse.ArgumentParser*), 868
parse_args() (*méthode optparse.OptionParser*), 911
PARSE_COLNAMES (*dans le module sqlite3*), 540
parse_config_h() (*dans le module sysconfig*), 1964
PARSE_DECLTYPES (*dans le module sqlite3*), 541
parse_headers() (*dans le module http.client*), 1458
parse_intermixed_args() (*méthode argparse.ArgumentParser*), 879
parse_known_args() (*méthode argparse.ArgumentParser*), 878
parse_known_intermixed_args() (*méthode argparse.ArgumentParser*), 879
parse_qs() (*dans le module urllib.parse*), 1445
parse_qsl() (*dans le module urllib.parse*), 1445
parseaddr() (*dans le module email.utils*), 1304
parsebytes() (*méthode email.parser.BytesParser*), 1260
parsedate() (*dans le module email.utils*), 1305
parsedate_to_datetime() (*dans le module email.utils*), 1305
parsedate_tz() (*dans le module email.utils*), 1305
ParseError (*classe dans xml.etree.ElementTree*), 1373
ParseFile() (*méthode xml.parsers.expat.xmlparser*), 1404
ParseFlags() (*dans le module imaplib*), 1475
parser (*attribut pathlib.PurePath*), 450
Parser (*classe dans email.parser*), 1260
ParserCreate() (*dans le module xml.parsers.expat*), 1403
ParseResult (*classe dans urllib.parse*), 1449
ParseResultBytes (*classe dans urllib.parse*), 1449
parsestr() (*méthode email.parser.Parser*), 1260
parseString() (*dans le module xml.dom.minidom*), 1385
parseString() (*dans le module xml.dom.pulldom*), 1390
parseString() (*dans le module xml.sax*), 1391
parsing
    URL, 1443
ParsingError, 639
partial (*attribut asyncio.IncompleteReadError*), 1105
partial() (*dans le module functools*), 430
partial() (*méthode imaplib.IMAP4*), 1479
partialmethod (*classe dans functools*), 431
parties (*attribut asyncio.Barrier*), 1097
parties (*attribut threading.Barrier*), 968
partition() (*méthode bytearray*), 70
partition() (*méthode bytes*), 70
partition() (*méthode str*), 58
partitioned (*attribut http.cookies.Morsel*), 1510
parts (*attribut pathlib.PurePath*), 450
Pass (*classe dans ast*), 2117
pass_() (*méthode poplib.POP3*), 1472
patch() (*dans le module test.support*), 1854
patch() (*dans le module unittest.mock*), 1803
patch.dict() (*dans le module unittest.mock*), 1807
patch.multiple() (*dans le module unittest.mock*), 1808
patch.object() (*dans le module unittest.mock*), 1806
patch.stopall() (*dans le module unittest.mock*), 1810
PATH, 471, 502, 712, 713, 718, 719, 728, 1032, 1413, 1917, 2050
path
    configuration file, 2050
    module search, 496, 1946, 2049
    operations, 445, 472
path (*attribut http.cookiejar.Cookie*), 1519
path (*attribut http.cookies.Morsel*), 1510
path (*attribut http.server.BaseHTTPRequestHandler*), 1503
path (*attribut ImportError*), 109
path (*attribut importlib.abc.FileLoader*), 2071
path (*attribut importlib.machinery.AppleFrameworkLoader*), 2079
path (*attribut importlib.machinery.ExtensionFileLoader*), 2077
path (*attribut importlib.machinery.FileFinder*), 2076
path (*attribut importlib.machinery.SourceFileLoader*), 2076
path (*attribut importlib.machinery.SourcelessFileLoader*), 2077
path (*attribut os.DirEntry*), 696
Path (*classe dans pathlib*), 457
Path (*classe dans zipfile*), 590
path (*dans le module sys*), 1946
Path browser, 1659
path() (*dans le module importlib.resources*), 2087
path_hook() (*méthode de la classe importlib.machinery.FileFinder*), 2076
path_hooks (*dans le module sys*), 1946
path_importer_cache (*dans le module sys*), 1946
path_mtime() (*méthode importlib.abc.SourceLoader*), 2072
path_return_ok() (*méthode http.cookiejar.CookiePolicy*), 1517

path_stats() (*méthode importlib.abc.SourceLoader*), 2072
path_stats() (*méthode importlib.machinery.SourceFileLoader*), 2076
pathconf() (*dans le module os*), 693
pathconf_names (*dans le module os*), 693
PathEntryFinder (*classe dans importlib.abc*), 2069
PATHEXT, 502
PathFinder (*classe dans importlib.machinery*), 2075
PathInfo (*classe dans pathlib.types*), 472
pathlib
    module, 445
pathlib.types
    module, 472
PathLike (*classe dans os*), 666
pathname2url() (*dans le module urllib.request*), 1428
pathsep (*dans le module os*), 728
Path.stem (*dans le module zipfile*), 591
Path.suffix (*dans le module zipfile*), 591
Path.suffixes (*dans le module zipfile*), 591
--pattern
    option de ligne de commande unittest-discover, 1755
pattern (*attribut re.Pattern*), 147
pattern (*attribut re.PatternError*), 146
Pattern (*classe dans re*), 146
Pattern (*classe dans typing*), 1720
PatternError, 145
pause() (*dans le module signal*), 1238
pause_reading() (*méthode asyncio.ReadTransport*), 1136
pause_writing() (*méthode asyncio.BaseProtocol*), 1140
PAX_FORMAT (*dans le module tarfile*), 599
pax_headers (*attribut tarfile.TarFile*), 603
pax_headers (*attribut tarfile.TarInfo*), 604
pbkdf2_hmac() (*dans le module hashlib*), 649
pd() (*dans le module turtle*), 1585
pdb
    module, 1876
Pdb (*class in pdb*), 1876
Pdb (*classe dans pdb*), 1878
.pdbrc
    file, 1880
pdf() (*méthode statistics.NormalDist*), 405
peek() (*méthode bz2.BZ2File*), 575
peek() (*méthode gzip.GzipFile*), 572
peek() (*méthode io.BufferedReader*), 738
peek() (*méthode lzma.LZMAFile*), 580
peek() (*méthode weakref.finalize*), 295
PEM_cert_to_DER_cert() (*dans le module ssl*), 1194
pen() (*dans le module turtle*), 1585
pencolor() (*dans le module turtle*), 1586
pending (*attribut ssl.MemoryBIO*), 1221
pending() (*méthode ssl.SSLSocket*), 1205
PendingDeprecationWarning, 116
pendown() (*dans le module turtle*), 1585
pensize() (*dans le module turtle*), 1585
penup() (*dans le module turtle*), 1585
PEP, **2241**
PERCENT (*dans le module token*), 2144
PERCENTEQUAL (*dans le module token*), 2145
perf_counter() (*dans le module time*), 747
perf_counter_ns() (*dans le module time*), 747
Performance, 1893
perm() (*dans le module math*), 340
PermissionError, 116
permutations() (*dans le module itertools*), 417
persistence, 509
persistent
    objects, 509
persistent_id (*pickle protocol*), 517
persistent_id() (*méthode pickle.Pickler*), 512
persistent_load (*pickle protocol*), 517
persistent_load() (*méthode pickle.Unpickler*), 514
PF_CAN (*dans le module socket*), 1167
PF_DIVERT (*dans le module socket*), 1168
PF_PACKET (*dans le module socket*), 1168
PF_RDS (*dans le module socket*), 1169
pformat() (*dans le module pprint*), 308
pformat() (*méthode pprint.PrettyPrinter*), 309
pgettext() (*dans le module gettext*), 1554
pgettext() (*méthode gettext.GNUTranslations*), 1557
pgettext() (*méthode gettext.NullTranslations*), 1556
PGO (*dans le module test.support*), 1848
phase() (*dans le module cmath*), 348
pi (*dans le module cmath*), 350
pi (*dans le module math*), 346
pi() (*méthode xml.etree.ElementTree.TreeBuilder*), 1371
pickle
    module, 306, 509, 525, 526, 529
pickle() (*dans le module copyreg*), 525
PickleBuffer (*classe dans pickle*), 514
PickleError, 512
Pickler (*classe dans pickle*), 512
pickletools
    module, 2181
pickling
    objects, 509
PicklingError, 512
pid (*attribut asyncio.subprocess.Process*), 1101
pid (*attribut multiprocessing.Process*), 977
pid (*attribut subprocess.Popen*), 1039
PIDFD_NONBLOCK (*dans le module os*), 716
pidfd_open() (*dans le module os*), 716
pidfd_send_signal() (*dans le module signal*), 1239
PIPE (*dans le module subprocess*), 1030
Pipe() (*dans le module multiprocessing*), 980
pipe() (*dans le module os*), 679
pipe2() (*dans le module os*), 679
PIPE_BUF (*dans le module select*), 1226
pipe_connection_lost() (*méthode asyncio.SubprocessProtocol*), 1142
pipe_data_received() (*méthode asyncio.SubprocessProtocol*), 1142

PIPE_MAX_SIZE (*dans le module test.support*), 1848
pipes
    module, 2224
pkgutil
    module, 2059
placeholder (*attribut textwrap.TextWrapper*), 172
Placeholder (*dans le module functools*), 430
platform
    module, 800
platform (*dans le module sys*), 1946
platform() (*dans le module platform*), 800
platlibdir (*dans le module sys*), 1947
PlaySound() (*dans le module winsound*), 2194
plist
    file, 642
plistlib
    module, 642
plock() (*dans le module os*), 716
PLUS (*dans le module token*), 2144
plus() (*méthode decimal.Context*), 368
PLUSEQUAL (*dans le module token*), 2145
pm() (*dans le module pdb*), 1878
point d'entrée pour la recherche dans path, **2241**
POINTER() (*dans le module ctypes*), 841
pointer() (*dans le module ctypes*), 841
polar() (*dans le module cmath*), 348
Policy (*classe dans email.policy*), 1266
poll() (*dans le module select*), 1225
poll() (*méthode multiprocessing.connection.Connection*), 985
poll() (*méthode select.devpoll*), 1226
poll() (*méthode select.epoll*), 1228
poll() (*méthode select.poll*), 1228
poll() (*méthode subprocess.Popen*), 1037
PollSelector (*classe dans selectors*), 1233
poly() (*dans le module turtle*), 1593
Pool (*classe dans multiprocessing.pool*), 998
pop() (*méthode array.array*), 291
pop() (*méthode collections.deque*), 265
pop() (*méthode dict*), 92
pop() (*méthode frozenset*), 90
pop() (*méthode mailbox.Mailbox*), 1320
pop() (*sequence method*), 49
POP3
    protocol, 1471
POP3 (*classe dans poplib*), 1471
POP3_SSL (*classe dans poplib*), 1471
pop_all() (*méthode contextlib.ExitStack*), 1997
POP_BLOCK (*opcode*), 2180
POP_EXCEPT (*opcode*), 2169
POP_JUMP_IF_FALSE (*opcode*), 2174
POP_JUMP_IF_NONE (*opcode*), 2174
POP_JUMP_IF_NOT_NONE (*opcode*), 2174
POP_JUMP_IF_TRUE (*opcode*), 2174
pop_source() (*méthode shlex.shlex*), 1615
POP_TOP (*opcode*), 2166
Popen (*classe dans subprocess*), 1032
popen() (*dans le module os*), 716
popen() (*in module os*), 1225
popitem() (*méthode collections.OrderedDict*), 273
popitem() (*méthode dict*), 92
popitem() (*méthode mailbox.Mailbox*), 1320
popleft() (*méthode collections.deque*), 265
poplib
    module, 1471
port (*attribut http.cookiejar.Cookie*), 1519
port_specified (*attribut http.cookiejar.Cookie*), 1520
portée imbriquée, **2240**
portion, **2241**
pos (*attribut json.JSONDecodeError*), 1314
pos (*attribut re.Match*), 150
pos (*attribut re.PatternError*), 146
pos (*attribut tomllib.TOMLDecodeError*), 640
pos() (*dans le module operator*), 439
pos() (*dans le module turtle*), 1583
position (*attribut xml.etree.ElementTree.ParseError*), 1374
position() (*dans le module turtle*), 1583
positions (*attribut inspect.FrameInfo*), 2039
positions (*attribut inspect.Traceback*), 2039
Positions (*classe dans dis*), 2165
Positions.col_offset (*dans le module dis*), 2165
Positions.end_col_offset (*dans le module dis*), 2165
Positions.end_lineno (*dans le module dis*), 2165
Positions.lineno (*dans le module dis*), 2165
POSIX
    I/O control, 2200
    threads, 1056
posix
    module, 2197
POSIX Shared Memory, 1014
POSIX_FADV_DONTNEED (*dans le module os*), 679
POSIX_FADV_NOREUSE (*dans le module os*), 679
POSIX_FADV_NORMAL (*dans le module os*), 679
POSIX_FADV_RANDOM (*dans le module os*), 679
POSIX_FADV_SEQUENTIAL (*dans le module os*), 679
POSIX_FADV_WILLNEED (*dans le module os*), 679
posix_fadvise() (*dans le module os*), 679
posix_fallocate() (*dans le module os*), 679
posix_openpt() (*dans le module os*), 680
posix_spawn() (*dans le module os*), 717
POSIX_SPAWN_CLOSE (*dans le module os*), 717
POSIX_SPAWN_CLOSEFROM (*dans le module os*), 717
POSIX_SPAWN_DUP2 (*dans le module os*), 717
POSIX_SPAWN_OPEN (*dans le module os*), 717
posix_spawnp() (*dans le module os*), 718
PosixPath (*classe dans pathlib*), 457
post_handshake_auth (*attribut ssl.SSLContext*), 1213
post_mortem() (*dans le module pdb*), 1878
post_setup() (*méthode venv.EnvBuilder*), 1919
postcmd() (*méthode cmd.Cmd*), 1609
postloop() (*méthode cmd.Cmd*), 1609
Pow (*classe dans ast*), 2110
pow()

built-in function, 26
pow() (*dans le module math*), 344
pow() (*dans le module operator*), 439
power() (*méthode decimal.Context*), 368
pp (*pdb command*), 1882
pp() (*dans le module pprint*), 307
pprint
module, 307
pprint() (*dans le module pprint*), 308
pprint() (*méthode pprint.PrettyPrinter*), 309
prcal() (*dans le module calendar*), 255
pread() (*dans le module os*), 679
preadv() (*dans le module os*), 680
--preamble
option de ligne de commande pickletools, 2182
preamble (*attribut email.message.EmailMessage*), 1258
preamble (*attribut email.message.Message*), 1295
precmd() (*méthode cmd.Cmd*), 1609
prefix (*attribut xml.dom.Attr*), 1381
prefix (*attribut xml.dom.Node*), 1377
prefix (*attribut zipimport.zipimporter*), 2059
prefix (*dans le module sys*), 1947
PREFIXES (*dans le module site*), 2051
prefixlen (*attribut ipaddress.IPv4Network*), 1541
prefixlen (*attribut ipaddress.IPv6Network*), 1544
preloop() (*méthode cmd.Cmd*), 1609
prepare() (*méthode graphlib.TopologicalSorter*), 332
prepare() (*méthode logging.handlers.QueueHandler*), 798
prepare() (*méthode logging.handlers.QueueListener*), 799
prepare_class() (*dans le module types*), 300
prepare_input_source() (*dans le module xml.sax.saxutils*), 1399
PrepareProtocol (*classe dans sqlite3*), 556
PrettyPrinter (*classe dans pprint*), 308
prev() (*méthode tkinter.ttk.Treeview*), 1653
previousSibling (*attribut xml.dom.Node*), 1376
print()
built-in function, 26
print() (*méthode traceback.TracebackException*), 2014
print_call_graph() (*dans le module asyncio*), 1106
print_callees() (*méthode pstats.Stats*), 1891
print_callers() (*méthode pstats.Stats*), 1891
print_exc() (*dans le module traceback*), 2011
print_exc() (*méthode timeit.Timer*), 1896
print_exception() (*dans le module traceback*), 2011
print_help() (*méthode argparse.ArgumentParser*), 877
print_last() (*dans le module traceback*), 2011
print_list() (*dans le module traceback*), 2012
print_stack() (*dans le module traceback*), 2011
print_stack() (*méthode asyncio.Task*), 1082
print_stats() (*méthode profile.Profile*), 1889
print_stats() (*méthode pstats.Stats*), 1891
print_tb() (*dans le module traceback*), 2011
print_usage() (*méthode argparse.ArgumentParser*), 877
print_usage() (*méthode optparse.OptionParser*), 913
print_version() (*méthode optparse.OptionParser*), 903
print_warning() (*dans le module test.support*), 1851
printable (*dans le module string*), 122
printdir() (*méthode zipfile.ZipFile*), 588
printf-style formatting, 63, 79
PRIO_DARWIN_BG (*dans le module os*), 668
PRIO_DARWIN_NONUI (*dans le module os*), 668
PRIO_DARWIN_PROCESS (*dans le module os*), 668
PRIO_DARWIN_THREAD (*dans le module os*), 668
PRIO_PGRP (*dans le module os*), 668
PRIO_PROCESS (*dans le module os*), 668
PRIO_USER (*dans le module os*), 668
PriorityQueue (*classe dans asyncio*), 1103
PriorityQueue (*classe dans queue*), 1049
prlimit() (*dans le module resource*), 2207
prmonth() (*dans le module calendar*), 254
prmonth() (*méthode calendar.TextCalendar*), 252
ProactorEventLoop (*classe dans asyncio*), 1128
process
group, 667, 668
id, 668
id of parent, 668
killing, 716
scheduling priority, 668, 671
signalling, 716
--process
option de ligne de commande timeit, 1896
Process (*classe dans multiprocessing*), 976
process() (*méthode logging.LoggerAdapter*), 769
process_cpu_count() (*dans le module os*), 727
process_exited() (*méthode asyncio.SubprocessProtocol*), 1142
process_request() (*méthode socketserver.BaseServer*), 1497
process_time() (*dans le module time*), 747
process_time_ns() (*dans le module time*), 747
process_tokens() (*dans le module tabnanny*), 2152
ProcessError, 979
processes, light-weight, 1056
ProcessingInstruction() (*dans le module xml.etree.ElementTree*), 1363
processingInstruction() (*méthode xml.sax.handler.ContentHandler*), 1396
ProcessingInstructionHandler() (*méthode xml.parsers.expat.xmlparser*), 1407
ProcessLookupError, 116
processor time, 747, 752
processor() (*dans le module platform*), 801
ProcessPoolExecutor (*classe dans concurrent.futures*), 1024
prod() (*dans le module math*), 344
product() (*dans le module itertools*), 417
profile

module, 1888
Profile (*classe dans profile*), 1888
profile function, 957, 1941, 1948
profiler, 1941, 1948
profiling, deterministic, 1885
ProgrammingError, 556
Progressbar (*classe dans tkinter.ttk*), 1648
prompt (*attribut cmd.Cmd*), 1609
prompts, interpreter, 1948
propagate (*attribut logging.Logger*), 757
property (*classe de base*), 26
property list, 642
property() (*dans le module enum*), 329
property_declaration_handler (*dans le module xml.sax.handler*), 1394
property_dom_node (*dans le module xml.sax.handler*), 1394
property_lexical_handler (*dans le module xml.sax.handler*), 1394
property_xml_string (*dans le module xml.sax.handler*), 1394
property.deleter()
built-in function, 27
property.getter()
built-in function, 27
PropertyMock (*classe dans unittest.mock*), 1795
property.setter()
built-in function, 27
prot_c() (*méthode ftplib.FTP_TLS*), 1470
prot_p() (*méthode ftplib.FTP_TLS*), 1470
proto (*attribut socket.socket*), 1185
protocol
context management, 95
copy, 516
FTP, 1442, 1465
HTTP, 1442, 1454, 1457, 1502
IMAP4, 1474
IMAP4_SSL, 1474
IMAP4_stream, 1474
iterator, 46
POP3, 1471
SMTP, 1482
protocol (*attribut ssl.SSLContext*), 1213
Protocol (*classe dans asyncio*), 1139
Protocol (*classe dans typing*), 1704
PROTOCOL_SSLv3 (*dans le module ssl*), 1197
PROTOCOL_SSLv23 (*dans le module ssl*), 1197
PROTOCOL_TLS (*dans le module ssl*), 1196
PROTOCOL_TLS_CLIENT (*dans le module ssl*), 1196
PROTOCOL_TLS_SERVER (*dans le module ssl*), 1197
PROTOCOL_TLSv1 (*dans le module ssl*), 1197
PROTOCOL_TLSv1_1 (*dans le module ssl*), 1197
PROTOCOL_TLSv1_2 (*dans le module ssl*), 1197
protocol_version (*attribut http.server.BaseHTTPRequestHandler*), 1504
PROTOCOL_VERSION (*attribut imaplib.IMAP4*), 1481
ProtocolError (*classe dans xmlrpc.client*), 1526
proxy() (*dans le module weakref*), 293
proxyauth() (*méthode imaplib.IMAP4*), 1479
ProxyBasicAuthHandler (*classe dans urllib.request*), 1430
ProxyDigestAuthHandler (*classe dans urllib.request*), 1431
ProxyHandler (*classe dans urllib.request*), 1430
ProxyType (*dans le module weakref*), 296
ProxyTypes (*dans le module weakref*), 296
pryear() (*méthode calendar.TextCalendar*), 252
ps1 (*dans le module sys*), 1948
ps2 (*dans le module sys*), 1948
pstats
module, 1889
pstdev() (*dans le module statistics*), 400
pthread_getcpuclockid() (*dans le module time*), 745
pthread_kill() (*dans le module signal*), 1239
pthread_sigmask() (*dans le module signal*), 1239
pthreads, 1056
pthreads (*attribut sys._emscripten_info*), 1933
ptsname() (*dans le module os*), 680
pty
module, 679, 2202
pu() (*dans le module turtle*), 1585
publicId (*attribut xml.dom.DocumentType*), 1378
PullDom (*classe dans xml.dom.pulldom*), 1390
punctuation (*dans le module string*), 122
punctuation_chars (*attribut shlex.shlex*), 1617
PurePath (*classe dans pathlib*), 447
PurePosixPath (*classe dans pathlib*), 448
PureWindowsPath (*classe dans pathlib*), 448
purge() (*dans le module re*), 145
Purpose.CLIENT_AUTH (*dans le module ssl*), 1201
Purpose.SERVER_AUTH (*dans le module ssl*), 1201
push() (*méthode code.InteractiveConsole*), 2055
push() (*méthode contextlib.ExitStack*), 1997
push_async_callback() (*méthode contextlib.AsyncExitStack*), 1998
push_async_exit() (*méthode contextlib.AsyncExitStack*), 1998
PUSH_EXC_INFO (*opcode*), 2169
PUSH_NULL (*opcode*), 2176
push_source() (*méthode shlex.shlex*), 1615
push_token() (*méthode shlex.shlex*), 1615
put() (*méthode asyncio.Queue*), 1103
put() (*méthode multiprocessing.Queue*), 980
put() (*méthode multiprocessing.SimpleQueue*), 982
put() (*méthode queue.Queue*), 1050
put() (*méthode queue.SimpleQueue*), 1052
put_nowait() (*méthode asyncio.Queue*), 1103
put_nowait() (*méthode multiprocessing.Queue*), 981
put_nowait() (*méthode queue.Queue*), 1050
put_nowait() (*méthode queue.SimpleQueue*), 1052
putch() (*dans le module msvcrt*), 2184
putenv() (*dans le module os*), 669
putheader() (*méthode http.client.HTTPConnection*), 1462

putp() (*dans le module curses*), 928
putrequest() (*méthode http.client.HTTPConnection*), 1461
putwch() (*dans le module msvcrt*), 2184
putwin() (*méthode curses.window*), 935
pvariance() (*dans le module statistics*), 400
pwd
    module, 474, 2198
pwd() (*méthode ftplib.FTP*), 1469
pwrite() (*dans le module os*), 681
pwritev() (*dans le module os*), 681
py_compile
    module, 2154
Py_DEBUG (*dans le module test.support*), 1848
py_object (*classe dans ctypes*), 845
PY_RESUME (*monitoring event*), 1956
PY_RETURN (*monitoring event*), 1956
PY_START (*monitoring event*), 1956
PY_THROW (*monitoring event*), 1956
PY_UNWIND (*monitoring event*), 1956
PY_YIELD (*monitoring event*), 1956
pyc utilisant le hachage, **2236**
pycache_prefix (*dans le module sys*), 1933
PyCF_ALLOW_TOP_LEVEL_AWAIT (*dans le module ast*), 2138
PyCF_ONLY_AST (*dans le module ast*), 2138
PyCF_OPTIMIZED_AST (*dans le module ast*), 2138
PyCF_TYPE_COMMENTS (*dans le module ast*), 2138
PycInvalidationMode (*classe dans py_compile*), 2155
pyclbr
    module, 2152
PyCompileError, 2154
PyDLL (*classe dans ctypes*), 833
pydoc
    module, 1724
pyexpat
    module, 1403
PYFUNCTYPE() (*dans le module ctypes*), 837
--python
    option de ligne de commande zipapp, 1925
Python 3000, **2241**
Python Editor, 1659
Python Enhancement Proposals
    PEP 1, 2241
    PEP 8, 29
    PEP 205, 296
    PEP 227, 2020
    PEP 235, 2066
    PEP 236, 2021
    PEP 237, 64, 80
    PEP 238, 2020
    PEP 246, 556
    PEP 249, 536, 539, 551, 556, 559, 565
    PEP 0249#threadsafety, 542
    PEP 255, 2020
    PEP 263, 2066, 2148, 2149
    PEP 273, 2058
    PEP 278, 2244
    PEP 282, 504, 774
    PEP 292, 130
    PEP 302, 34, 496, 1946, 2058, 2060, 2061, 2064, 2066, 20692071, 2238
    PEP 305, 613
    PEP 307, 510
    PEP 324, 1028
    PEP 328, 34, 2020, 2066, 2234
    PEP 338, 2065
    PEP 342, 279
    PEP 343, 2001, 2020, 2232
    PEP 362, 2036, 2230, 2241
    PEP 366, 2065, 2066
    PEP 370, 2052
    PEP 378, 126
    PEP 380#use-of-stopiteration-to-return-values, 1958
    PEP 383, 192, 1163
    PEP 387, 116
    PEP 393, 198, 1945
    PEP 405, 1915
    PEP 411, 1942, 1950, 1951, 2241
    PEP 412, 427
    PEP 420, 2066, 2240, 2241
    PEP 421, 1944
    PEP 428, 446
    PEP 434, 1670
    PEP 442, 2025
    PEP 443, 2235
    PEP 451, 1946, 20642066
    PEP 453, 1913
    PEP 461, 80
    PEP 468, 274
    PEP 475, 25, 115, 677, 682, 684, 722, 748, 1178, 1179, 11811184, 12261229, 1233, 1241
    PEP 479, 112, 2020
    PEP 483, 2236
    PEP 484, 100, 1673, 1683, 1696, 1713, 2106, 2134, 2138, 2229, 2235, 2236, 2244
    PEP 485, 342, 350
    PEP 488, 1862, 2066, 2080, 2154
    PEP 489, 2066, 2075, 2077, 2081
    PEP 492, 280, 2043, 2230, 2232
    PEP 495, 245
    PEP 498, 2234
    PEP 506, 659
    PEP 515, 126, 381
    PEP 519, 2241
    PEP 524, 729
    PEP 525, 280, 1942, 1950, 2043, 2230
    PEP 526, 1689, 1704, 1978, 1986, 2045, 2046, 2134, 2138, 2229, 2244
    PEP 529, 689, 1940, 1951
    PEP 538, 1567
    PEP 540, 664, 1567
    PEP 544, 1683, 1705

PEP 552, 2066, 2155
PEP 557, 1978
PEP 560, 300, 301
PEP 563, 1718, 2020, 2045, 2046
PEP 565, 116
PEP 566, 2093
PEP 567, 1052, 1110, 1111, 1133
PEP 574, 511, 523
PEP 578, 1865, 1930
PEP 584, 260, 268, 274, 294, 304, 666
PEP 585, 100, 277, 303, 17171724, 2236
PEP 586, 1688
PEP 589, 1709
PEP 591, 1689, 1714
PEP 593, 1692, 1716
PEP 594, 2217, 22212225
PEP 597, 732
PEP 604, 101
PEP 610, 2095
PEP 612, 1675, 1681, 1688, 1701, 1723
PEP 613, 1686
PEP 615, 245
PEP 626, 2164
PEP 632, 2222
PEP 644, 1190
PEP 646, 1699
PEP 647, 1693
PEP 649, 2020, 2021, 2045, 2046, 2229
PEP 649#the-stringizer-and-the-fake-globals-environment, 2047
PEP 655, 1689, 1709
PEP 667, 22, 1878
PEP 673, 1686
PEP 675, 1684
PEP 681, 1713
PEP 682, 126
PEP 683, 2236
PEP 686, 665, 731
PEP 688, 280
PEP 692, 1694
PEP 695, 1684, 1696, 1697, 1699, 1701, 1724
PEP 698, 1715
PEP 702, 1978
PEP 703, 2234, 2236
PEP 705, 1690
PEP 706, 605
PEP 709, 22
PEP 734, 1023, 1024
PEP 742, 1693
PEP 749, 2021, 2045
PEP 3101, 122, 123
PEP 3105, 2020
PEP 3107, 2045, 2046
PEP 3112, 2020
PEP 3115, 300, 2132
PEP 3116, 2244
PEP 3118, 82, 511
PEP 3119, 281, 2004
PEP 3120, 2066
PEP 3134, 108
PEP 3141, 335, 2004
PEP 3147, 1862, 2064, 2066, 2080, 2154, 2156, 2158
PEP 3148, 1027
PEP 3149, 1929
PEP 3151, 116, 1165, 1224, 2206
PEP 3154, 511
PEP 3155, 2242
PEP 3333, 14161420, 1423, 1424
PEP 3333#input-and-error-streams, 1424
PEP 3333#optional-platform-specific-file-handling, 1424
PEP 3333#the-start-response-callable, 1424
python_branch() (*dans le module platform*), 801
python_build() (*dans le module platform*), 801
python_compiler() (*dans le module platform*), 801
PYTHON_CPU_COUNT, 727, 983
PYTHON_DOM, 1375
PYTHON_GIL, 2236
python_implementation() (*dans le module platform*), 801
python_is_optimized() (*dans le module test.support*), 1850
python_revision() (*dans le module platform*), 801
python_version() (*dans le module platform*), 801
python_version_tuple() (*dans le module platform*), 801
PYTHONASYNCIODEBUG, 1124, 1159, 1726
PYTHONBREAKPOINT, 10, 1932
PYTHONCASEOK, 34
PYTHONCOERCECLOCALE, 665
PYTHONDEVMODE, 1725
PYTHONDONDONTWRITEBYTECODE, 1933
PYTHONFAULTHANDLER, 1726, 1874
PythonFinalizationError, 111
PYTHONHOME, 1857, 2097, 2098
PYTHONINTMAXSTRDIGITS, 105, 1944
PYTHONIOENCODING, 664, 1952
Pythonique, **2242**
PYTHONLEGACYWINDOWSFSENCODING, 1951
PYTHONLEGACYWINDOWSSTDIO, 1952
PYTHONMALLOC, 1726
PYTHONNOUSERSITE, 2051
PYTHONPATH, 1857, 1946, 2097
PYTHONPLATLIBDIR, 2098
PYTHONPYCACHEPREFIX, 1933
PYTHONSAFEPATH, 1946, 2227
PYTHONSTARTUP, 178, 1062, 1666, 1944, 2051
PYTHONTRACEMALLOC, 1901, 1906
PYTHONTZPATH, 250
PYTHONUNBUFFERED, 1952
PYTHONUSERBASE, 2051
PYTHONUSERSITE, 1857
PYTHONUTF8, 664, 665, 1952
PYTHONWARNDEFAULTENCODING, 732

PYTHONWARNINGS, 1726, 1972, 1973
PyZipFile (*classe dans zipfile*), 592

Q

-q
    option de ligne de commande compileall, 2156
    option de ligne de commande python--m-py_compile, 2155
qiflush() (*dans le module curses*), 928
QName (*classe dans xml.etree.ElementTree*), 1370
qsize() (*méthode asyncio.Queue*), 1103
qsize() (*méthode multiprocessing.Queue*), 980
qsize() (*méthode queue.Queue*), 1050
qsize() (*méthode queue.SimpleQueue*), 1052
quantiles() (*dans le module statistics*), 401
quantiles() (*méthode statistics.NormalDist*), 405
quantize() (*méthode decimal.Context*), 368
quantize() (*méthode decimal.Decimal*), 361
QueryInfoKey() (*dans le module winreg*), 2189
QueryReflectionKey() (*dans le module winreg*), 2191
QueryValue() (*dans le module winreg*), 2189
QueryValueEx() (*dans le module winreg*), 2189
QUESTION (*dans le module tkinter.messagebox*), 1639
queue
    module, 1049
queue (*attribut sched.scheduler*), 1048
Queue (*classe dans asyncio*), 1102
Queue (*classe dans multiprocessing*), 980
Queue (*classe dans queue*), 1049
Queue() (*méthode multiprocessing.managers.SyncManager*), 993
QueueEmpty, 1104
QueueFull, 1104
QueueHandler (*classe dans logging.handlers*), 798
QueueListener (*classe dans logging.handlers*), 799
QueueShutDown, 1104
quick_ratio() (*méthode difflib.SequenceMatcher*), 162
--quiet
    option de ligne de commande python--m-py_compile, 2155
quiet (*attribut sys.flags*), 1936
quit (*pdb command*), 1884
quit (*variable de base*), 36
quit() (*méthode ftplib.FTP*), 1469
quit() (*méthode poplib.POP3*), 1473
quit() (*méthode smtplib.SMTP*), 1487
quit() (*méthode tkinter.filedialog.FileDialog*), 1636
quitting (*bdb.Bdb attribute*), 1872
quopri
    module, 1345
quote() (*dans le module email.utils*), 1304
quote() (*dans le module shlex*), 1613
quote() (*dans le module urllib.parse*), 1450
QUOTE_ALL (*dans le module csv*), 617
quote_from_bytes() (*dans le module urllib.parse*), 1450
QUOTE_MINIMAL (*dans le module csv*), 617
QUOTE_NONE (*dans le module csv*), 617
QUOTE_NONNUMERIC (*dans le module csv*), 617
QUOTE_NOTNULL (*dans le module csv*), 617
quote_plus() (*dans le module urllib.parse*), 1450
QUOTE_STRINGS (*dans le module csv*), 617
quoteattr() (*dans le module xml.sax.saxutils*), 1398
quotechar (*attribut csv.Dialect*), 618
quoted-printable
    encoding, 1345
quotes (*attribut shlex.shlex*), 1616
quoting (*attribut csv.Dialect*), 618

R

-R
    option de ligne de commande trace, 1899
-r
    option de ligne de commande compileall, 2156
    option de ligne de commande timeit, 1896
    option de ligne de commande trace, 1899
R_OK (*dans le module os*), 687
radians() (*dans le module math*), 345
radians() (*dans le module turtle*), 1585
radix (*attribut sys.float_info*), 1938
radix() (*méthode decimal.Context*), 368
radix() (*méthode decimal.Decimal*), 361
RADIXCHAR (*dans le module locale*), 1564
raise
    statement, 107
Raise (*classe dans ast*), 2117
RAISE (*monitoring event*), 1956
raise_on_defect (*attribut email.policy.Policy*), 1267
raise_signal() (*dans le module signal*), 1238
RAISE_VARARGS (*opcode*), 2176
raiseExceptions (*dans le module logging*), 773
ramasse-miettes, **2235**
RAND_add() (*dans le module ssl*), 1193
RAND_bytes() (*dans le module ssl*), 1193
RAND_status() (*dans le module ssl*), 1193
randbelow() (*dans le module secrets*), 659
randbits() (*dans le module secrets*), 659
randbytes() (*dans le module random*), 385
randint() (*dans le module random*), 385
random
    module, 383
Random (*classe dans random*), 388
random() (*dans le module random*), 387
random() (*méthode random.Random*), 388
randrange() (*dans le module random*), 385
range
    object, 51
range (*classe de base*), 51

RARROW (*dans le module token*), 2146
ratio() (*méthode difflib.SequenceMatcher*), 162
Rational (*classe dans numbers*), 336
raw (*attribut io.BufferedIOBase*), 736
raw() (*dans le module curses*), 928
raw() (*méthode pickle.PickleBuffer*), 514
raw_data_manager (*dans le module email.contentmanager*), 1279
raw_decode() (*méthode json.JSONDecoder*), 1312
raw_input() (*méthode code.InteractiveConsole*), 2055
RawArray() (*dans le module multiprocessing.sharedctypes*), 989
RawConfigParser (*classe dans configparser*), 637
RawDescriptionHelpFormatter (*classe dans argparse*), 853
RawIOBase (*classe dans io*), 735
RawPen (*classe dans turtle*), 1602
RawTextHelpFormatter (*classe dans argparse*), 853
RawTurtle (*classe dans turtle*), 1602
RawValue() (*dans le module multiprocessing.sharedctypes*), 989
RBRACE (*dans le module token*), 2145
re
    module, 54, 132, 495
re (*attribut re.Match*), 150
READ (*attribut inspect.BufferFlags*), 2044
read() (*dans le module os*), 681
read() (*méthode asyncio.StreamReader*), 1087
read() (*méthode codecs.StreamReader*), 197
read() (*méthode configparser.ConfigParser*), 635
read() (*méthode http.client.HTTPResponse*), 1462
read() (*méthode imaplib.IMAP4*), 1479
read() (*méthode io.BufferedIOBase*), 736
read() (*méthode io.BufferedReader*), 738
read() (*méthode io.RawIOBase*), 735
read() (*méthode io.TextIOBase*), 740
read() (*méthode mimetypes.MimeTypes*), 1338
read() (*méthode mmap.mmap*), 1246
read() (*méthode sqlite3.Blob*), 555
read() (*méthode ssl.MemoryBIO*), 1221
read() (*méthode ssl.SSLSocket*), 1202
read() (*méthode urllib.robotparser.RobotFileParser*), 1453
read() (*méthode zipfile.ZipFile*), 589
read1() (*méthode bz2.BZ2File*), 576
read1() (*méthode io.BufferedIOBase*), 736
read1() (*méthode io.BufferedReader*), 739
read1() (*méthode io.BytesIO*), 738
read_binary() (*dans le module importlib.resources*), 2087
read_byte() (*méthode mmap.mmap*), 1246
read_bytes() (*méthode importlib.abc.Traversable*), 2074
read_bytes() (*méthode importlib.resources.abc.Traversable*), 2089
read_bytes() (*méthode pathlib.Path*), 463
read_bytes() (*méthode zipfile.Path*), 591
read_dict() (*méthode configparser.ConfigParser*), 636
read_environ() (*dans le module wsgiref.handlers*), 1424
read_events() (*méthode xml.etree.ElementTree.XMLPullParser*), 1373
read_file() (*méthode configparser.ConfigParser*), 636
read_history_file() (*dans le module readline*), 176
read_init_file() (*dans le module readline*), 176
read_mime_types() (*dans le module mimetypes*), 1337
read_string() (*méthode configparser.ConfigParser*), 636
read_text() (*dans le module importlib.resources*), 2087
read_text() (*méthode importlib.abc.Traversable*), 2074
read_text() (*méthode importlib.resources.abc.Traversable*), 2089
read_text() (*méthode pathlib.Path*), 462
read_text() (*méthode zipfile.Path*), 591
read_token() (*méthode shlex.shlex*), 1615
read_windows_registry() (*méthode mimetypes.MimeTypes*), 1339
READABLE (*dans le module _tkinter*), 1632
readable() (*méthode bz2.BZ2File*), 575
readable() (*méthode io.IOBase*), 734
readall() (*méthode io.RawIOBase*), 735
reader() (*dans le module csv*), 613
ReadError, 597
readexactly() (*méthode asyncio.StreamReader*), 1087
readfp() (*méthode mimetypes.MimeTypes*), 1339
readframes() (*méthode wave.Wave_read*), 1550
readinto() (*dans le module os*), 682
readinto() (*méthode bz2.BZ2File*), 576
readinto() (*méthode http.client.HTTPResponse*), 1462
readinto() (*méthode io.BufferedIOBase*), 736
readinto() (*méthode io.RawIOBase*), 735
readinto1() (*méthode io.BufferedIOBase*), 737
readinto1() (*méthode io.BytesIO*), 738
readline
    module, 175
readline() (*méthode asyncio.StreamReader*), 1087
readline() (*méthode codecs.StreamReader*), 197
readline() (*méthode imaplib.IMAP4*), 1479
readline() (*méthode io.IOBase*), 734
readline() (*méthode io.TextIOBase*), 740
readline() (*méthode mmap.mmap*), 1246
readlines() (*méthode codecs.StreamReader*), 197
readlines() (*méthode io.IOBase*), 734
readlink() (*dans le module os*), 693
readlink() (*méthode pathlib.Path*), 460
readmodule() (*dans le module pyclbr*), 2152
readmodule_ex() (*dans le module pyclbr*), 2152
readonly (*attribut memoryview*), 87
ReadOnly (*dans le module typing*), 1689
ReadTransport (*classe dans asyncio*), 1135
readuntil() (*méthode asyncio.StreamReader*), 1087

readv() (*dans le module os*), 684
ready() (*méthode multiprocessing.pool.AsyncResult*), 1000
real (*attribut numbers.Complex*), 335
Real (*classe dans numbers*), 335
real_max_memuse (*dans le module test.support*), 1848
real_quick_ratio() (*méthode difflib.SequenceMatcher*), 162
realpath() (*dans le module os.path*), 476
REALTIME_PRIORITY_CLASS (*dans le module subprocess*), 1041
reap_children() (*dans le module test.support*), 1853
reap_threads() (*dans le module test.support.threading_helper*), 1858
reason (*attribut http.client.HTTPResponse*), 1463
reason (*attribut ssl.SSLError*), 1192
reason (*attribut UnicodeError*), 114
reason (*attribut urllib.error.HTTPError*), 1452
reason (*attribut urllib.error.URLError*), 1452
reattach() (*méthode tkinter.ttk.Treeview*), 1653
recent() (*méthode imaplib.IMAP4*), 1479
reconfigure() (*méthode io.TextIOWrapper*), 741
record_original_stdout() (*dans le module test.support*), 1850
RECORDS (*attribut inspect.BufferFlags*), 2044
records (*attribut unittest.TestCase*), 1766
RECORDS_RO (*attribut inspect.BufferFlags*), 2044
rect() (*dans le module cmath*), 348
rectangle() (*dans le module curses.textpad*), 948
RecursionError, 112
recursive_repr() (*dans le module reprlib*), 313
recv() (*méthode multiprocessing.connection.Connection*), 984
recv() (*méthode socket.socket*), 1181
recv_bytes() (*méthode multiprocessing.connection.Connection*), 985
recv_bytes_into() (*méthode multiprocessing.connection.Connection*), 985
recv_fds() (*dans le module socket*), 1178
recv_into() (*méthode socket.socket*), 1183
recvfrom() (*méthode socket.socket*), 1181
recvfrom_into() (*méthode socket.socket*), 1182
recvmsg() (*méthode socket.socket*), 1181
recvmsg_into() (*méthode socket.socket*), 1182
redirect_request() (*méthode urllib.request.HTTPRedirectHandler*), 1435
redirect_stderr() (*dans le module contextlib*), 1994
redirect_stdout() (*dans le module contextlib*), 1993
redisplay() (*dans le module readline*), 176
redrawln() (*méthode curses.window*), 935
redrawwin() (*méthode curses.window*), 936
reduce() (*dans le module functools*), 432
reducer_override() (*méthode pickle.Pickler*), 513
ref (*classe dans weakref*), 293
refcount_test() (*dans le module test.support*), 1853
référence empruntée, **2231**
référence forte, **2243**
ReferenceError, 112
ReferenceType (*dans le module weakref*), 296
refold_source (*attribut email.policy.EmailPolicy*), 1269
refresh() (*méthode curses.window*), 936
REG_BINARY (*dans le module winreg*), 2192
REG_DWORD (*dans le module winreg*), 2192
REG_DWORD_BIG_ENDIAN (*dans le module winreg*), 2193
REG_DWORD_LITTLE_ENDIAN (*dans le module winreg*), 2193
REG_EXPAND_SZ (*dans le module winreg*), 2193
REG_FULL_RESOURCE_DESCRIPTOR (*dans le module winreg*), 2193
REG_LINK (*dans le module winreg*), 2193
REG_MULTI_SZ (*dans le module winreg*), 2193
REG_NONE (*dans le module winreg*), 2193
REG_QWORD (*dans le module winreg*), 2193
REG_QWORD_LITTLE_ENDIAN (*dans le module winreg*), 2193
REG_RESOURCE_LIST (*dans le module winreg*), 2193
REG_RESOURCE_REQUIREMENTS_LIST (*dans le module winreg*), 2193
REG_SZ (*dans le module winreg*), 2193
RegexFlag (*classe dans re*), 140
register() (*dans le module atexit*), 2009
register() (*dans le module codecs*), 190
register() (*dans le module faulthandler*), 1875
register() (*dans le module webbrowser*), 1414
register() (*méthode abc.ABCMeta*), 2004
register() (*méthode argparse.ArgumentParser*), 879
register() (*méthode multiprocessing.managers.BaseManager*), 992
register() (*méthode select.devpoll*), 1226
register() (*méthode select.epoll*), 1227
register() (*méthode selectors.BaseSelector*), 1232
register() (*méthode select.poll*), 1228
register_adapter() (*dans le module sqlite3*), 540
register_archive_format() (*dans le module shutil*), 504
register_at_fork() (*dans le module os*), 718
register_callback() (*dans le module sys.monitoring*), 1959
register_converter() (*dans le module sqlite3*), 540
register_defect() (*méthode email.policy.Policy*), 1267
register_dialect() (*dans le module csv*), 614
register_error() (*dans le module codecs*), 193
register_function() (*méthode xmlrpc.server.CGIXMLRPCRequestHandler*), 1532
register_function() (*méthode xmlrpc.server.SimpleXMLRPCServer*), 1529
register_instance() (*méthode xmlrpc.server.CGIXMLRPCRequestHandler*), 1532
register_instance() (*méthode xmlrpc.server.SimpleXMLRPCServer*), 1529
register_introspection_functions() (*méthode*

xmlrpc.server.CGIXMLRPCRequestHandler), 1533
register_introspection_functions() (*méthode xmlrpc.server.SimpleXMLRPCServer*), 1530
register_multicall_functions() (*méthode xmlrpc.server.CGIXMLRPCRequestHandler*), 1533
register_multicall_functions() (*méthode xmlrpc.server.SimpleXMLRPCServer*), 1530
register_namespace() (*dans le module xml.etree.ElementTree*), 1363
register_optionflag() (*dans le module doctest*), 1737
register_shape() (*dans le module turtle*), 1600
register_unpack_format() (*dans le module shutil*), 505
registerDOMImplementation() (*dans le module xml.dom*), 1375
registerResult() (*dans le module unittest*), 1783
REGTYPE (*dans le module tarfile*), 598
relative
URL, 1443
relative_to() (*méthode pathlib.PurePath*), 455
release() (*dans le module platform*), 801
release() (*méthode _thread.lock*), 1058
release() (*méthode asyncio.Condition*), 1095
release() (*méthode asyncio.Lock*), 1093
release() (*méthode asyncio.Semaphore*), 1096
release() (*méthode logging.Handler*), 762
release() (*méthode memoryview*), 84
release() (*méthode multiprocessing.Lock*), 987
release() (*méthode multiprocessing.RLock*), 988
release() (*méthode pickle.PickleBuffer*), 514
release() (*méthode threading.Condition*), 964
release() (*méthode threading.Lock*), 962
release() (*méthode threading.RLock*), 963
release() (*méthode threading.Semaphore*), 965
reload() (*dans le module importlib*), 2067
reload_environ() (*dans le module os*), 666
relpath() (*dans le module os.path*), 477
remainder() (*dans le module math*), 341
remainder() (*méthode decimal.Context*), 369
remainder_near() (*méthode decimal.Context*), 369
remainder_near() (*méthode decimal.Decimal*), 361
RemoteDisconnected, 1459
remove() (*dans le module os*), 694
remove() (*méthode array.array*), 291
remove() (*méthode collections.deque*), 265
remove() (*méthode frozenset*), 90
remove() (*méthode mailbox.Mailbox*), 1318
remove() (*méthode mailbox.MH*), 1324
remove() (*méthode xml.etree.ElementTree.Element*), 1368
remove() (*sequence method*), 49
remove_done_callback() (*méthode asyncio.Future*), 1133
remove_done_callback() (*méthode asyncio.Task*), 1082
remove_flag() (*méthode mailbox.Maildir*), 1322
remove_flag() (*méthode mailbox.MaildirMessage*), 1328
remove_flag() (*méthode mailbox.mboxMessage*), 1329
remove_flag() (*méthode mailbox.MMDFMessage*), 1333
remove_folder() (*méthode mailbox.Maildir*), 1321
remove_folder() (*méthode mailbox.MH*), 1324
remove_header() (*méthode urllib.request.Request*), 1432
remove_history_item() (*dans le module readline*), 177
remove_label() (*méthode mailbox.BabylMessage*), 1332
remove_option() (*méthode configparser.ConfigParser*), 637
remove_option() (*méthode optparse.OptionParser*), 912
remove_reader() (*méthode asyncio.loop*), 1118
remove_section() (*méthode configparser.ConfigParser*), 637
remove_sequence() (*méthode mailbox.MHMessage*), 1331
remove_signal_handler() (*méthode asyncio.loop*), 1121
remove_writer() (*méthode asyncio.loop*), 1118
removeAttribute() (*méthode xml.dom.Element*), 1380
removeAttributeNode() (*méthode xml.dom.Element*), 1380
removeAttributeNS() (*méthode xml.dom.Element*), 1380
removeChild() (*méthode xml.dom.Node*), 1378
removedirs() (*dans le module os*), 694
removeFilter() (*méthode logging.Handler*), 763
removeFilter() (*méthode logging.Logger*), 761
removeHandler() (*dans le module unittest*), 1783
removeHandler() (*méthode logging.Logger*), 761
removeprefix() (*méthode bytearray*), 68
removeprefix() (*méthode bytes*), 68
removeprefix() (*méthode str*), 59
removeResult() (*dans le module unittest*), 1783
removesuffix() (*méthode bytearray*), 68
removesuffix() (*méthode bytes*), 68
removesuffix() (*méthode str*), 59
removexattr() (*dans le module os*), 711
rename() (*dans le module os*), 694
rename() (*méthode ftplib.FTP*), 1468
rename() (*méthode imaplib.IMAP4*), 1479
rename() (*méthode pathlib.Path*), 468
renames() (*dans le module os*), 694
reopenIfNeeded() (*méthode logging.handlers.WatchedFileHandler*), 788
reorganize() (*méthode dbm.gnu.gdbm*), 534
--repeat
option de ligne de commande timeit, 1896

repeat() (*dans le module itertools*), 418
repeat() (*dans le module timeit*), 1894
repeat() (*méthode timeit.Timer*), 1895
repetition
    operation, 47
REPL, **2242**
replace
    error handler's name, 192
replace() (*dans le module copy*), 306
replace() (*dans le module dataclasses*), 1984
replace() (*dans le module os*), 695
replace() (*méthode bytearray*), 70
replace() (*méthode bytes*), 70
replace() (*méthode curses.panel.Panel*), 954
replace() (*méthode datetime.date*), 215
replace() (*méthode datetime.datetime*), 223
replace() (*méthode datetime.time*), 231
replace() (*méthode inspect.Parameter*), 2035
replace() (*méthode inspect.Signature*), 2033
replace() (*méthode pathlib.Path*), 468
replace() (*méthode str*), 59
replace() (*méthode tarfile.TarInfo*), 604
replace_errors() (*dans le module codecs*), 193
replace_header() (*méthode email.message.EmailMessage*), 1253
replace_header() (*méthode email.message.Message*), 1292
replace_history_item() (*dans le module readline*), 177
replace_whitespace (*attribut textwrap.TextWrapper*), 170
replaceChild() (*méthode xml.dom.Node*), 1378
ReplacePackage() (*dans le module modulefinder*), 2062
--report
    option de ligne de commande trace, 1899
report() (*méthode filecmp.dircmp*), 485
report() (*méthode modulefinder.ModuleFinder*), 2062
REPORT_CDIFF (*dans le module doctest*), 1737
REPORT_ERRMODE (*dans le module msvcrt*), 2185
report_failure() (*méthode doctest.DocTestRunner*), 1746
report_full_closure() (*méthode filecmp.dircmp*), 485
REPORT_NDIFF (*dans le module doctest*), 1737
REPORT_ONLY_FIRST_FAILURE (*dans le module doctest*), 1737
report_partial_closure() (*méthode filecmp.dircmp*), 485
report_start() (*méthode doctest.DocTestRunner*), 1746
report_success() (*méthode doctest.DocTestRunner*), 1746
REPORT_UDIFF (*dans le module doctest*), 1736
report_unexpected_exception() (*méthode doctest.DocTestRunner*), 1746
REPORTING_FLAGS (*dans le module doctest*), 1737
Repr (*classe dans reprlib*), 313
repr()
    built-in function, 28
repr() (*dans le module reprlib*), 313
repr() (*méthode reprlib.Repr*), 315
repr1() (*méthode reprlib.Repr*), 315
ReprEnum (*classe dans enum*), 326
reprlib
    module, 313
request (*attribut socketserver.BaseRequestHandler*), 1498
Request (*classe dans urllib.request*), 1428
request() (*méthode http.client.HTTPConnection*), 1459
request_queue_size (*attribut socketserver.BaseServer*), 1497
request_rate() (*méthode urllib.robotparser.RobotFileParser*), 1453
request_uri() (*dans le module wsgiref.util*), 1416
request_version (*attribut http.server.BaseHTTPRequestHandler*), 1503
RequestHandlerClass (*attribut socketserver.BaseServer*), 1497
requestline (*attribut http.server.BaseHTTPRequestHandler*), 1503
Required (*dans le module typing*), 1689
requires() (*dans le module importlib.metadata*), 2094
requires() (*dans le module test.support*), 1850
requires_bz2() (*dans le module test.support*), 1853
requires_docstrings() (*dans le module test.support*), 1853
requires_freebsd_version() (*dans le module test.support*), 1852
requires_gil_enabled() (*dans le module test.support*), 1852
requires_gzip() (*dans le module test.support*), 1852
requires_IEEE_754() (*dans le module test.support*), 1852
requires_limited_api() (*dans le module test.support*), 1853
requires_linux_version() (*dans le module test.support*), 1852
requires_lzma() (*dans le module test.support*), 1853
requires_mac_version() (*dans le module test.support*), 1852
requires_resource() (*dans le module test.support*), 1853
requires_zlib() (*dans le module test.support*), 1852
RERAISE (*monitoring event*), 1956
RERAISE (*opcode*), 2169
reschedule() (*méthode asyncio.Timeout*), 1075
reserved (*attribut zipfile.ZipInfo*), 594
RESERVED_FUTURE (*dans le module uuid*), 1491
RESERVED_MICROSOFT (*dans le module uuid*), 1491
RESERVED_NCS (*dans le module uuid*), 1491
reset() (*dans le module turtle*), 1589

reset() (*méthode asyncio.Barrier*), 1097
reset() (*méthode bdb.Bdb*), 1870
reset() (*méthode codecs.IncrementalDecoder*), 195
reset() (*méthode codecs.IncrementalEncoder*), 195
reset() (*méthode codecs.StreamReader*), 197
reset() (*méthode codecs.StreamWriter*), 196
reset() (*méthode contextvars.ContextVar*), 1053
reset() (*méthode html.parser.HTMLParser*), 1349
reset() (*méthode threading.Barrier*), 968
reset() (*méthode xml.dom.pulldom.DOMEventStream*), 1390
reset() (*méthode xml.sax.xmlreader.IncrementalParser*), 1401
reset_mock() (*méthode unittest.mock.AsyncMock*), 1798
reset_mock() (*méthode unittest.mock.Mock*), 1788
reset_peak() (*dans le module tracemalloc*), 1906
reset_prog_mode() (*dans le module curses*), 928
reset_shell_mode() (*dans le module curses*), 928
reset_tzpath() (*dans le module zoneinfo*), 249
resetbuffer() (*méthode code.InteractiveConsole*), 2055
resetscreen() (*dans le module turtle*), 1596
resetty() (*dans le module curses*), 928
resetwarnings() (*dans le module warnings*), 1977
resize() (*dans le module ctypes*), 841
resize() (*méthode bytearray*), 67
resize() (*méthode curses.window*), 936
resize() (*méthode mmap.mmap*), 1246
resize_term() (*dans le module curses*), 928
resizemode() (*dans le module turtle*), 1590
resizeterm() (*dans le module curses*), 928
resolution (*attribut datetime.date*), 214
resolution (*attribut datetime.datetime*), 221
resolution (*attribut datetime.time*), 229
resolution (*attribut datetime.timedelta*), 210
resolve() (*méthode pathlib.Path*), 459
resolve_bases() (*dans le module types*), 300
resolve_name() (*dans le module importlib.util*), 2080
resolve_name() (*dans le module pkgutil*), 2061
resolveEntity() (*méthode xml.sax.handler.EntityResolver*), 1397
resource
    module, 2206
resource_path() (*méthode importlib.abc.ResourceReader*), 2073
resource_path() (*méthode importlib.resources.abc.ResourceReader*), 2088
ResourceDenied, 1847
ResourceLoader (*classe dans importlib.abc*), 2070
ResourceReader (*classe dans importlib.abc*), 2072
ResourceReader (*classe dans importlib.resources.abc*), 2088
ResourceWarning, 117
response() (*méthode imaplib.IMAP4*), 1479
ResponseNotReady, 1459
responses (*attribut http.server.BaseHTTPRequestHandler*), 1504
responses (*dans le module http.client*), 1459
restart (*pdb command*), 1884
restart_events() (*dans le module sys.monitoring*), 1958
restore() (*dans le module difflib*), 159
restore() (*méthode test.support.SaveSignals*), 1856
restype (*attribut ctypes._CFuncPtr*), 836
result() (*méthode asyncio.Future*), 1132
result() (*méthode asyncio.Task*), 1081
result() (*méthode concurrent.futures.Future*), 1026
results() (*méthode trace.Trace*), 1900
RESUME (*opcode*), 2178
resume_reading() (*méthode asyncio.ReadTransport*), 1137
resume_writing() (*méthode asyncio.BaseProtocol*), 1140
retours à la ligne universels, **2244**
retr() (*méthode poplib.POP3*), 1473
retrbinary() (*méthode ftplib.FTP*), 1467
retrlines() (*méthode ftplib.FTP*), 1467
RETRY (*dans le module tkinter.messagebox*), 1638
RETRYCANCEL (*dans le module tkinter.messagebox*), 1639
Return (*classe dans ast*), 2131
return (*pdb command*), 1881
return_annotation (*attribut inspect.Signature*), 2033
RETURN_GENERATOR (*opcode*), 2178
return_ok() (*méthode http.cookiejar.CookiePolicy*), 1516
return_value (*attribut unittest.mock.Mock*), 1790
RETURN_VALUE (*opcode*), 2169
returncode (*attribut asyncio.subprocess.Process*), 1101
returncode (*attribut subprocess.CalledProcessError*), 1031
returncode (*attribut subprocess.CompletedProcess*), 1030
returncode (*attribut subprocess.Popen*), 1039
retval (*pdb command*), 1884
reveal_type() (*dans le module typing*), 1711
reverse() (*méthode array.array*), 291
reverse() (*méthode collections.deque*), 265
reverse() (*sequence method*), 49
reverse_order() (*méthode pstats.Stats*), 1891
reverse_pointer (*attribut ipaddress.IPv4Address*), 1536
reverse_pointer (*attribut ipaddress.IPv6Address*), 1538
reversed()
    built-in function, 28
Reversible (*classe dans collections.abc*), 279
Reversible (*classe dans typing*), 1723
revert() (*méthode http.cookiejar.FileCookieJar*), 1515
rewind() (*méthode wave.Wave_read*), 1550
RFC
    RFC 821, 1482, 1484
    RFC 822, 750, 1281, 1299, 1462, 14851487, 1556
    RFC 959, 1465
    RFC 1123, 750

RFC 1321, 645
RFC 1422, 1215, 1223
RFC 1521, 1342, 1345
RFC 1522, 1344, 1345
RFC 1730, 1474
RFC 1738, 1451
RFC 1750, 1194
RFC 1766, 1565
RFC 1808, 1443, 1444, 1451
RFC 1869, 1482, 1484
RFC 1939, 1471
RFC 2045, 1249, 1254, 1276, 1292, 1293, 1299, 1339, 1342
RFC 2045 Section 6.8, 1524
RFC 2046, 1249, 1280, 1299
RFC 2047, 1249, 1269, 1274, 1299, 1300, 1305
RFC 2060, 1474, 1480
RFC 2068, 1509
RFC 2104, 657
RFC 2109, 15091514, 15181520
RFC 2177, 1477
RFC 2183, 1249, 1255, 1295
RFC 2231, 1249, 1253, 1254, 1292, 1293, 1299, 1306
RFC 2295, 1455
RFC 2324, 1455
RFC 2342, 1478
RFC 2368, 1451
RFC 2373, 1536, 1537
RFC 2396, 1446, 1450, 1451
RFC 2397, 1438
RFC 2449, 1472
RFC 2518, 1454
RFC 2595, 1471, 1473
RFC 2616, 1417, 1420, 1435, 1452
RFC 2616 Section 5.1.2, 1460
RFC 2616 Section 14.23, 1460
RFC 2640, 1465, 1466, 1470
RFC 2732, 1451
RFC 2774, 1455
RFC 2821, 1249
RFC 2822, 750, 1291, 1299, 1300, 1304, 1305, 1326, 1458, 1503
RFC 2964, 1514
RFC 2965, 1429, 1432, 15121514, 15161520
RFC 3056, 1538
RFC 3171, 1536
RFC 3229, 1455
RFC 3280, 1203
RFC 3330, 1537
RFC 3454, 174
RFC 3490, 203205
RFC 3490 Section 3.1, 205
RFC 3492, 203, 204
RFC 3493, 1189
RFC 3501, 1480
RFC 3542, 1176
RFC 3548, 1343
RFC 3659, 1468
RFC 3879, 1538
RFC 3927, 1537
RFC 3986, 1443, 1445, 1447, 1448, 1450, 1451, 1503
RFC 4007, 1537, 1538
RFC 4086, 1223
RFC 4122, 1488, 1490, 1491
RFC 4180, 613
RFC 4193, 1538
RFC 4217, 1469
RFC 4291, 1537
RFC 4380, 1538
RFC 4627, 1307, 1315
RFC 4648, 1339, 1340, 1342, 2227
RFC 4918, 1455
RFC 4954, 1485
RFC 5161, 1476
RFC 5246, 1201, 1224
RFC 5280, 1191, 1192, 1194, 1223
RFC 5321, 1278
RFC 5322, 1249, 1250, 1260, 1263, 1264, 1266, 1268, 1269, 12721275, 1278, 1288, 1487
RFC 5424, 794
RFC 5735, 1537
RFC 5789, 1457
RFC 5842, 1455
RFC 5891, 204
RFC 5895, 204
RFC 5929, 1204
RFC 6066, 1199, 1209, 1224
RFC 6531, 1251, 1269, 1482
RFC 6532, 1249, 1250, 1260, 1269
RFC 6585, 1455
RFC 6855, 1476
RFC 6856, 1473
RFC 7159, 1307, 1314, 1315
RFC 7230, 1428, 1462
RFC 7301, 1199, 1209
RFC 7525, 1224
RFC 7693, 650
RFC 7725, 1455
RFC 7914, 649
RFC 8089, 458
RFC 8297, 1454
RFC 8305, 1113
RFC 8470, 1455
RFC 9110, 14541457
RFC 9562, 14881492
RFC 9562 Section 5.8, 1491
RFC 9562 Section 5.9, 1492
RFC 9562 Section 5.10, 1492
rfc2109 (*attribut http.cookiejar.Cookie*), 1519
rfc2109_as_netscape (*attribut http.cookiejar.DefaultCookiePolicy*), 1518
rfc2965 (*attribut http.cookiejar.CookiePolicy*), 1517
RFC_4122 (*dans le module uuid*), 1491

rfile (*attribut http.server.BaseHTTPRequestHandler*), 1503
rfile (*attribut socketserver.DatagramRequestHandler*), 1498
rfind() (*méthode bytearray*), 70
rfind() (*méthode bytes*), 70
rfind() (*méthode mmap.mmap*), 1247
rfind() (*méthode str*), 59
rgb_to_hls() (*dans le module colorsys*), 1552
rgb_to_hsv() (*dans le module colorsys*), 1552
rgb_to_yiq() (*dans le module colorsys*), 1552
rglob() (*méthode pathlib.Path*), 464
right (*attribut filecmp.dircmp*), 486
right() (*dans le module turtle*), 1579
right_list (*attribut filecmp.dircmp*), 486
right_only (*attribut filecmp.dircmp*), 486
RIGHTSHIFT (*dans le module token*), 2145
RIGHTSHIFTEQUAL (*dans le module token*), 2145
rindex() (*méthode bytearray*), 70
rindex() (*méthode bytes*), 70
rindex() (*méthode str*), 59
rjust() (*méthode bytearray*), 72
rjust() (*méthode bytes*), 72
rjust() (*méthode str*), 59
rlcompleter
    module, 180
RLIM_INFINITY (*dans le module resource*), 2207
RLIMIT_AS (*dans le module resource*), 2208
RLIMIT_CORE (*dans le module resource*), 2207
RLIMIT_CPU (*dans le module resource*), 2207
RLIMIT_DATA (*dans le module resource*), 2208
RLIMIT_FSIZE (*dans le module resource*), 2207
RLIMIT_KQUEUES (*dans le module resource*), 2209
RLIMIT_MEMLOCK (*dans le module resource*), 2208
RLIMIT_MSGQUEUE (*dans le module resource*), 2208
RLIMIT_NICE (*dans le module resource*), 2208
RLIMIT_NOFILE (*dans le module resource*), 2208
RLIMIT_NPROC (*dans le module resource*), 2208
RLIMIT_NPTS (*dans le module resource*), 2209
RLIMIT_OFILE (*dans le module resource*), 2208
RLIMIT_RSS (*dans le module resource*), 2208
RLIMIT_RTPRIO (*dans le module resource*), 2208
RLIMIT_RTTIME (*dans le module resource*), 2208
RLIMIT_SBSIZE (*dans le module resource*), 2209
RLIMIT_SIGPENDING (*dans le module resource*), 2208
RLIMIT_STACK (*dans le module resource*), 2208
RLIMIT_SWAP (*dans le module resource*), 2209
RLIMIT_VMEM (*dans le module resource*), 2208
RLock (*classe dans multiprocessing*), 987
RLock (*classe dans threading*), 962
RLock() (*méthode multiprocessing.managers.SyncManager*), 993
rmd() (*méthode ftplib.FTP*), 1469
rmdir() (*dans le module os*), 695
rmdir() (*dans le module test.support.os_helper*), 1860
rmdir() (*méthode pathlib.Path*), 468
rmtree() (*dans le module shutil*), 500
rmtree() (*dans le module test.support.os_helper*), 1860
RobotFileParser (*classe dans urllib.robotparser*), 1452
robots.txt, 1452
rollback() (*méthode sqlite3.Connection*), 543
rollover() (*méthode tempfile.SpooledTemporaryFile*), 488
ROMAN (*dans le module tkinter.font*), 1633
root (*attribut pathlib.PurePath*), 450
rotate() (*méthode collections.deque*), 265
rotate() (*méthode decimal.Context*), 369
rotate() (*méthode decimal.Decimal*), 362
rotate() (*méthode logging.handlers.BaseRotatingHandler*), 789
RotatingFileHandler (*classe dans logging.handlers*), 789
rotation_filename() (*méthode logging.handlers.BaseRotatingHandler*), 789
rotator (*attribut logging.handlers.BaseRotatingHandler*), 789
round()
    built-in function, 28
ROUND_05UP (*dans le module decimal*), 370
ROUND_CEILING (*dans le module decimal*), 370
ROUND_DOWN (*dans le module decimal*), 370
ROUND_FLOOR (*dans le module decimal*), 370
ROUND_HALF_DOWN (*dans le module decimal*), 370
ROUND_HALF_EVEN (*dans le module decimal*), 370
ROUND_HALF_UP (*dans le module decimal*), 370
ROUND_UP (*dans le module decimal*), 370
Rounded (*classe dans decimal*), 371
rounds (*attribut sys.float_info*), 1938
Row (*classe dans sqlite3*), 555
row_factory (*attribut sqlite3.Connection*), 552
row_factory (*attribut sqlite3.Cursor*), 554
rowcount (*attribut sqlite3.Cursor*), 554
RPAR (*dans le module token*), 2144
rpartition() (*méthode bytearray*), 71
rpartition() (*méthode bytes*), 71
rpartition() (*méthode str*), 59
rpc_paths (*attribut xmlrpc.server.SimpleXMLRPCRequestHandler*), 1530
rpop() (*méthode poplib.POP3*), 1472
RS (*dans le module curses.ascii*), 951
rset() (*méthode poplib.POP3*), 1473
RShift (*classe dans ast*), 2110
rshift() (*dans le module operator*), 439
rsplit() (*méthode bytearray*), 72
rsplit() (*méthode bytes*), 72
rsplit() (*méthode str*), 59
RSQB (*dans le module token*), 2144
rstrip() (*méthode bytearray*), 72
rstrip() (*méthode bytes*), 72
rstrip() (*méthode str*), 59
rt() (*dans le module turtle*), 1579
RTLD_DEEPBIND (*dans le module os*), 728
RTLD_GLOBAL (*dans le module os*), 728
RTLD_LAZY (*dans le module os*), 728
RTLD_LOCAL (*dans le module os*), 728

RTLD_NODELETE (*dans le module os*), 728
RTLD_NOLOAD (*dans le module os*), 728
RTLD_NOW (*dans le module os*), 728
ruler (*attribut cmd.Cmd*), 1610
run (*pdb command*), 1884
Run script, 1661
run() (*dans le module asyncio*), 1062
run() (*dans le module pdb*), 1878
run() (*dans le module profile*), 1888
run() (*dans le module subprocess*), 1028
run() (*méthode asyncio.Runner*), 1063
run() (*méthode bdb.Bdb*), 1873
run() (*méthode contextvars.Context*), 1054
run() (*méthode doctest.DocTestRunner*), 1747
run() (*méthode multiprocessing.Process*), 976
run() (*méthode pdb.Pdb*), 1879
run() (*méthode profile.Profile*), 1889
run() (*méthode sched.scheduler*), 1048
run() (*méthode threading.Thread*), 959
run() (*méthode trace.Trace*), 1900
run() (*méthode unittest.IsolatedAsyncioTestCase*), 1772
run() (*méthode unittest.TestCase*), 1762
run() (*méthode unittest.TestSuite*), 1773
run() (*méthode unittest.TextTestRunner*), 1779
run() (*méthode wsgiref.handlers.BaseHandler*), 1422
run_coroutine_threadsafe() (*dans le module asyncio*), 1079
run_docstring_examples() (*dans le module doctest*), 1740
run_forever() (*méthode asyncio.loop*), 1109
run_in_executor() (*méthode asyncio.loop*), 1121
run_in_subinterp() (*dans le module test.support*), 1854
run_module() (*dans le module runpy*), 2064
run_path() (*dans le module runpy*), 2064
run_python_until_end() (*dans le module test.support.script_helper*), 1857
run_script() (*méthode modulefinder.ModuleFinder*), 2062
run_until_complete() (*méthode asyncio.loop*), 1109
run_with_locale() (*dans le module test.support*), 1852
run_with_tz() (*dans le module test.support*), 1852
runcall() (*dans le module pdb*), 1878
runcall() (*méthode bdb.Bdb*), 1873
runcall() (*méthode pdb.Pdb*), 1879
runcall() (*méthode profile.Profile*), 1889
runcode() (*méthode code.InteractiveInterpreter*), 2054
runctx() (*dans le module profile*), 1888
runctx() (*méthode bdb.Bdb*), 1873
runctx() (*méthode profile.Profile*), 1889
runctx() (*méthode trace.Trace*), 1900
runeval() (*dans le module pdb*), 1878
runeval() (*méthode bdb.Bdb*), 1873
runeval() (*méthode pdb.Pdb*), 1879
runfunc() (*méthode trace.Trace*), 1900
Runner (*classe dans asyncio*), 1063
running() (*méthode concurrent.futures.Future*), 1026
runpy
    module, 2063
runsource() (*méthode code.InteractiveInterpreter*), 2054
runtime (*attribut sys._emscripten_info*), 1933
runtime_checkable() (*dans le module typing*), 1705
RuntimeError, 112
RuntimeWarning, 116
RUSAGE_BOTH (*dans le module resource*), 2210
RUSAGE_CHILDREN (*dans le module resource*), 2210
RUSAGE_SELF (*dans le module resource*), 2210
RUSAGE_THREAD (*dans le module resource*), 2210
RWF_APPEND (*dans le module os*), 681
RWF_DSYNC (*dans le module os*), 681
RWF_HIPRI (*dans le module os*), 680
RWF_NOWAIT (*dans le module os*), 680
RWF_SYNC (*dans le module os*), 681

S

-S
    option de ligne de commande dis, 2161
-s
    option de ligne de commande calendar, 258
    option de ligne de commande compileall, 2156
    option de ligne de commande timeit, 1896
    option de ligne de commande trace, 1899
    option de ligne de commande unittest-discover, 1755
S (*dans le module re*), 141
S_ENFMT (*dans le module stat*), 482
S_IEXEC (*dans le module stat*), 482
S_IFBLK (*dans le module stat*), 481
S_IFCHR (*dans le module stat*), 481
S_IFDIR (*dans le module stat*), 481
S_IFDOOR (*dans le module stat*), 481
S_IFIFO (*dans le module stat*), 481
S_IFLNK (*dans le module stat*), 481
S_IFMT() (*dans le module stat*), 479
S_IFPORT (*dans le module stat*), 481
S_IFREG (*dans le module stat*), 481
S_IFSOCK (*dans le module stat*), 481
S_IFWHT (*dans le module stat*), 481
S_IMODE() (*dans le module stat*), 479
S_IREAD (*dans le module stat*), 482
S_IRGRP (*dans le module stat*), 482
S_IROTH (*dans le module stat*), 482
S_IRUSR (*dans le module stat*), 482
S_IRWXG (*dans le module stat*), 482
S_IRWXO (*dans le module stat*), 482
S_IRWXU (*dans le module stat*), 482
S_ISBLK() (*dans le module stat*), 479
S_ISCHR() (*dans le module stat*), 479
S_ISDIR() (*dans le module stat*), 479

S_ISDOOR() (*dans le module stat*), 479
S_ISFIFO() (*dans le module stat*), 479
S_ISGID (*dans le module stat*), 481
S_ISLNK() (*dans le module stat*), 479
S_ISPORT() (*dans le module stat*), 479
S_ISREG() (*dans le module stat*), 479
S_ISSOCK() (*dans le module stat*), 479
S_ISUID (*dans le module stat*), 481
S_ISVTX (*dans le module stat*), 482
S_ISWHT() (*dans le module stat*), 479
S_IWGRP (*dans le module stat*), 482
S_IWOTH (*dans le module stat*), 482
S_IWRITE (*dans le module stat*), 482
S_IWUSR (*dans le module stat*), 482
S_IXGRP (*dans le module stat*), 482
S_IXOTH (*dans le module stat*), 482
S_IXUSR (*dans le module stat*), 482
safe (*attribut uuid.SafeUUID*), 1488
safe_path (*attribut sys.flags*), 1936
safe_substitute() (*méthode string.Template*), 130
saferepr() (*dans le module pprint*), 308
SafeUUID (*classe dans uuid*), 1488
same_files (*attribut filecmp.dircmp*), 486
same_quantum() (*méthode decimal.Context*), 369
same_quantum() (*méthode decimal.Decimal*), 362
samefile() (*dans le module os.path*), 477
samefile() (*méthode pathlib.Path*), 462
SameFileError, 498
sameopenfile() (*dans le module os.path*), 477
samesite (*attribut http.cookies.Morsel*), 1510
samestat() (*dans le module os.path*), 477
sample() (*dans le module random*), 386
samples() (*méthode statistics.NormalDist*), 405
SATURDAY (*dans le module calendar*), 255
save() (*dans le module turtle*), 1601
save() (*méthode http.cookiejar.FileCookieJar*), 1515
save() (*méthode test.support.SaveSignals*), 1856
SaveAs (*classe dans tkinter.filedialog*), 1635
SAVEDCWD (*dans le module test.support.os_helper*), 1859
SaveFileDialog (*classe dans tkinter.filedialog*), 1636
SaveKey() (*dans le module winreg*), 2190
SaveSignals (*classe dans test.support*), 1856
savetty() (*dans le module curses*), 928
SAX2DOM (*classe dans xml.dom.pulldom*), 1390
SAXException, 1391
SAXNotRecognizedException, 1392
SAXNotSupportedException, 1392
SAXParseException, 1392
scaleb() (*méthode decimal.Context*), 369
scaleb() (*méthode decimal.Decimal*), 362
scandir() (*dans le module os*), 695
scanf (*C function*), 151
sched
    module, 1047
SCHED_BATCH (*dans le module os*), 725
SCHED_DEADLINE (*dans le module os*), 725
SCHED_FIFO (*dans le module os*), 726
sched_get_priority_max() (*dans le module os*), 726
sched_get_priority_min() (*dans le module os*), 726
sched_getaffinity() (*dans le module os*), 727
sched_getparam() (*dans le module os*), 726
sched_getscheduler() (*dans le module os*), 726
SCHED_IDLE (*dans le module os*), 726
SCHED_NORMAL (*dans le module os*), 726
SCHED_OTHER (*dans le module os*), 725
sched_param (*classe dans os*), 726
sched_priority (*attribut os.sched_param*), 726
SCHED_RESET_ON_FORK (*dans le module os*), 726
SCHED_RR (*dans le module os*), 726
sched_rr_get_interval() (*dans le module os*), 726
sched_setaffinity() (*dans le module os*), 727
sched_setparam() (*dans le module os*), 726
sched_setscheduler() (*dans le module os*), 726
SCHED_SPORADIC (*dans le module os*), 726
sched_yield() (*dans le module os*), 726
scheduler (*classe dans sched*), 1047
SCM_CREDS2 (*dans le module socket*), 1170
scope_id (*attribut ipaddress.IPv6Address*), 1538
Screen (*classe dans turtle*), 1602
screensize() (*dans le module turtle*), 1596
script_from_examples() (*dans le module doctest*), 1748
scroll() (*méthode curses.window*), 936
ScrolledCanvas (*classe dans turtle*), 1602
ScrolledText (*classe dans tkinter.scrolledtext*), 1639
scrollok() (*méthode curses.window*), 936
scrypt() (*dans le module hashlib*), 649
seal() (*dans le module unittest.mock*), 1823
search
    path, module, 496, 1946, 2049
search() (*dans le module re*), 142
search() (*méthode imaplib.IMAP4*), 1479
search() (*méthode re.Pattern*), 146
second (*attribut datetime.datetime*), 222
second (*attribut datetime.time*), 230
secondes depuis *epoch*, 744
seconds (*attribut datetime.timedelta*), 210
secrets
    module, 659
SECTCRE (*attribut configparser.ConfigParser*), 631
sections() (*méthode configparser.ConfigParser*), 635
secure (*attribut http.cookiejar.Cookie*), 1519
secure (*attribut http.cookies.Morsel*), 1510
secure hash algorithm, SHA1, SHA2, SHA224, SHA256, SHA384, SHA512, SHA3, Shake, Blake2, 645
Secure Sockets Layer, 1189
security
    http.server, 1508
security considerations, 2225
security_level (*attribut ssl.SSLContext*), 1213
see() (*méthode tkinter.ttk.Treeview*), 1653
seed() (*dans le module random*), 384

seed() (*méthode random.Random*), 388
seed_bits (*attribut sys.hash_info*), 1943
seek() (*méthode io.IOBase*), 735
seek() (*méthode io.TextIOBase*), 740
seek() (*méthode io.TextIOWrapper*), 742
seek() (*méthode mmap.mmap*), 1247
seek() (*méthode sqlite3.Blob*), 555
SEEK_CUR (*dans le module os*), 677
SEEK_DATA (*dans le module os*), 677
SEEK_END (*dans le module os*), 677
SEEK_HOLE (*dans le module os*), 677
SEEK_SET (*dans le module os*), 677
seekable() (*méthode bz2.BZ2File*), 575
seekable() (*méthode io.IOBase*), 735
seekable() (*méthode mmap.mmap*), 1247
select
    module, 1224
select() (*dans le module select*), 1225
select() (*méthode imaplib.IMAP4*), 1479
select() (*méthode selectors.BaseSelector*), 1232
select() (*méthode tkinter.ttk.Notebook*), 1647
selected_alpn_protocol() (*méthode ssl.SSLSocket*), 1205
selected_npn_protocol() (*méthode ssl.SSLSocket*), 1205
selection() (*méthode tkinter.ttk.Treeview*), 1653
selection_add() (*méthode tkinter.ttk.Treeview*), 1654
selection_remove() (*méthode tkinter.ttk.Treeview*), 1654
selection_set() (*méthode tkinter.ttk.Treeview*), 1654
selection_toggle() (*méthode tkinter.ttk.Treeview*), 1654
selector (*attribut urllib.request.Request*), 1432
SelectorEventLoop (*classe dans asyncio*), 1127
SelectorKey (*classe dans selectors*), 1232
selectors
    module, 1231
SelectSelector (*classe dans selectors*), 1233
Self (*dans le module typing*), 1685
Semaphore (*classe dans asyncio*), 1095
Semaphore (*classe dans multiprocessing*), 988
Semaphore (*classe dans threading*), 965
Semaphore() (*méthode multiprocessing.managers.SyncManager*), 993
semaphores, binary, 1056
SEMI (*dans le module token*), 2144
SEND (*opcode*), 2178
send() (*méthode http.client.HTTPConnection*), 1462
send() (*méthode imaplib.IMAP4*), 1479
send() (*méthode logging.handlers.DatagramHandler*), 793
send() (*méthode logging.handlers.SocketHandler*), 792
send() (*méthode multiprocessing.connection.Connection*), 984
send() (*méthode socket.socket*), 1183
send_bytes() (*méthode multiprocessing.connection.Connection*), 985
send_error() (*méthode http.server.BaseHTTPRequestHandler*), 1505
send_fds() (*dans le module socket*), 1178
send_header() (*méthode http.server.BaseHTTPRequestHandler*), 1505
send_message() (*méthode smtplib.SMTP*), 1487
send_response() (*méthode http.server.BaseHTTPRequestHandler*), 1505
send_response_only() (*méthode http.server.BaseHTTPRequestHandler*), 1505
send_signal() (*méthode asyncio.subprocess.Process*), 1100
send_signal() (*méthode asyncio.SubprocessTransport*), 1138
send_signal() (*méthode subprocess.Popen*), 1038
sendall() (*méthode socket.socket*), 1183
sendcmd() (*méthode ftplib.FTP*), 1467
sendfile() (*dans le module os*), 682
sendfile() (*méthode asyncio.loop*), 1117
sendfile() (*méthode socket.socket*), 1184
sendfile() (*méthode wsgiref.handlers.BaseHandler*), 1423
SendfileNotAvailableError, 1105
sendmail() (*méthode smtplib.SMTP*), 1486
sendmsg() (*méthode socket.socket*), 1183
sendmsg_afalg() (*méthode socket.socket*), 1184
sendto() (*méthode asyncio.DatagramTransport*), 1138
sendto() (*méthode socket.socket*), 1183
sentinel (*attribut multiprocessing.Process*), 978
sentinel (*dans le module unittest.mock*), 1815
sep (*dans le module os*), 728
SEPTEMBER (*dans le module calendar*), 256
sequence
    iteration, 46
    object, 47
    types, immutable, 48
    types, mutable, 49
    types, operations on, 47, 49
séquence, **2242**
Sequence (*classe dans collections.abc*), 279
Sequence (*classe dans typing*), 1721
SequenceMatcher (*classe dans difflib*), 160
serialize() (*méthode sqlite3.Connection*), 550
serializing
    objects, 509
serve_forever() (*méthode asyncio.Server*), 1127
serve_forever() (*méthode socketserver.BaseServer*), 1496
server
    WWW, 1502
server (*attribut http.server.BaseHTTPRequestHandler*), 1503

server (*attribut socketserver.BaseRequestHandler*), 1498
Server (*classe dans asyncio*), 1126
server_activate() (*méthode socketserver.BaseServer*), 1498
server_address (*attribut socketserver.BaseServer*), 1497
server_bind() (*méthode socketserver.BaseServer*), 1498
server_close() (*méthode socketserver.BaseServer*), 1496
server_hostname (*attribut ssl.SSLSocket*), 1205
server_side (*attribut ssl.SSLSocket*), 1205
server_software (*attribut wsgiref.handlers.BaseHandler*), 1422
server_version (*attribut http.server.BaseHTTPRequestHandler*), 1504
server_version (*attribut http.server.SimpleHTTPRequestHandler*), 1506
ServerProxy (*classe dans xmlrpc.client*), 1521
service_actions() (*méthode socketserver.BaseServer*), 1496
session (*attribut ssl.SSLSocket*), 1206
session_reused (*attribut ssl.SSLSocket*), 1206
session_stats() (*méthode ssl.SSLContext*), 1211
set
object, 88
Set (*classe dans ast*), 2108
Set (*classe dans collections.abc*), 279
Set (*classe dans typing*), 1719
set (*classe de base*), 88
set() (*méthode asyncio.Event*), 1093
set() (*méthode configparser.ConfigParser*), 637
set() (*méthode configparser.RawConfigParser*), 638
set() (*méthode contextvars.ContextVar*), 1053
set() (*méthode http.cookies.Morsel*), 1511
set() (*méthode test.support.os_helper.EnvironmentVarGuard*), 1860
set() (*méthode threading.Event*), 966
set() (*méthode tkinter.ttk.Combobox*), 1645
set() (*méthode tkinter.ttk.Spinbox*), 1646
set() (*méthode tkinter.ttk.Treeview*), 1654
set() (*méthode xml.etree.ElementTree.Element*), 1367
SET_ADD (*opcode*), 2168
set_allowed_domains() (*méthode http.cookiejar.DefaultCookiePolicy*), 1518
set_alpn_protocols() (*méthode ssl.SSLContext*), 1209
set_app() (*méthode wsgiref.simple_server.WSGIServer*), 1419
set_asyncgen_hooks() (*dans le module sys*), 1950
set_authorizer() (*méthode sqlite3.Connection*), 546
set_auto_history() (*dans le module readline*), 177
set_blocked_domains() (*méthode http.cookiejar.DefaultCookiePolicy*), 1518
set_blocking() (*dans le module os*), 683
set_boundary() (*méthode email.message.EmailMessage*), 1254
set_boundary() (*méthode email.message.Message*), 1294
set_break() (*méthode bdb.Bdb*), 1872
set_charset() (*méthode email.message.Message*), 1290
set_children() (*méthode tkinter.ttk.Treeview*), 1651
set_ciphers() (*méthode ssl.SSLContext*), 1209
set_completer() (*dans le module readline*), 178
set_completer_delims() (*dans le module readline*), 178
set_completion_display_matches_hook() (*dans le module readline*), 178
set_content() (*dans le module email.contentmanager*), 1280
set_content() (*méthode email.contentmanager.ContentManager*), 1279
set_content() (*méthode email.message.EmailMessage*), 1257
set_continue() (*méthode bdb.Bdb*), 1872
set_cookie() (*méthode http.cookiejar.CookieJar*), 1515
set_cookie_if_ok() (*méthode http.cookiejar.CookieJar*), 1515
set_coroutine_origin_tracking_depth() (*dans le module sys*), 1950
set_data() (*méthode importlib.abc.SourceLoader*), 2072
set_data() (*méthode importlib.machinery.SourceFileLoader*), 2076
set_date() (*méthode mailbox.MaildirMessage*), 1328
set_debug() (*dans le module gc*), 2022
set_debug() (*méthode asyncio.loop*), 1124
set_debuglevel() (*méthode ftplib.FTP*), 1466
set_debuglevel() (*méthode http.client.HTTPConnection*), 1461
set_debuglevel() (*méthode poplib.POP3*), 1472
set_debuglevel() (*méthode smtplib.SMTP*), 1484
set_default_executor() (*méthode asyncio.loop*), 1123
set_default_type() (*méthode email.message.EmailMessage*), 1254
set_default_type() (*méthode email.message.Message*), 1293
set_default_verify_paths() (*méthode ssl.SSLContext*), 1209
set_defaults() (*méthode argparse.ArgumentParser*), 877
set_defaults() (*méthode optparse.OptionParser*), 913
set_ecdh_curve() (*méthode ssl.SSLContext*), 1210
set_errno() (*dans le module ctypes*), 841
set_error_mode() (*dans le module msvcrt*), 2184
set_escdelay() (*dans le module curses*), 929
set_event_loop() (*dans le module asyncio*), 1108
set_event_loop() (*méthode asyn-*

cio.AbstractEventLoopPolicy), 1149
set_event_loop_policy() (*dans le module asyncio*), 1148
set_events() (*dans le module sys.monitoring*), 1958
set_exception() (*méthode asyncio.Future*), 1132
set_exception() (*méthode concurrent.futures.Future*), 1026
set_exception_handler() (*méthode asyncio.loop*), 1123
set_executable() (*dans le module multiprocessing*), 984
set_filter() (*méthode tkinter.filedialog.FileDialog*), 1636
set_flags() (*méthode mailbox.Maildir*), 1321
set_flags() (*méthode mailbox.MaildirMessage*), 1327
set_flags() (*méthode mailbox.mboxMessage*), 1329
set_flags() (*méthode mailbox.MMDFMessage*), 1333
set_forkserver_preload() (*dans le module multiprocessing*), 984
set_from() (*méthode mailbox.mboxMessage*), 1329
set_from() (*méthode mailbox.MMDFMessage*), 1333
SET_FUNCTION_ATTRIBUTE (*opcode*), 2177
set_handle_inheritable() (*dans le module os*), 686
set_history_length() (*dans le module readline*), 177
set_info() (*méthode mailbox.Maildir*), 1322
set_info() (*méthode mailbox.MaildirMessage*), 1328
set_inheritable() (*dans le module os*), 685
set_inheritable() (*méthode socket.socket*), 1184
set_int_max_str_digits() (*dans le module sys*), 1948
set_labels() (*méthode mailbox.BabylMessage*), 1332
set_last_error() (*dans le module ctypes*), 841
set_local_events() (*dans le module sys.monitoring*), 1958
set_memlimit() (*dans le module test.support*), 1850
set_name() (*méthode asyncio.Task*), 1082
set_next() (*méthode bdb.Bdb*), 1872
set_nonstandard_attr() (*méthode http.cookiejar.Cookie*), 1520
set_npn_protocols() (*méthode ssl.SSLContext*), 1209
set_ok() (*méthode http.cookiejar.CookiePolicy*), 1516
set_param() (*méthode email.message.EmailMessage*), 1254
set_param() (*méthode email.message.Message*), 1293
set_pasv() (*méthode ftplib.FTP*), 1467
set_payload() (*méthode email.message.Message*), 1290
set_policy() (*méthode http.cookiejar.CookieJar*), 1514
set_pre_input_hook() (*dans le module readline*), 177
set_progress_handler() (*méthode sqlite3.Connection*), 547
set_protocol() (*méthode asyncio.BaseTransport*), 1136
set_proxy() (*méthode urllib.request.Request*), 1432
set_psk_client_callback() (*méthode ssl.SSLContext*), 1213
set_psk_server_callback() (*méthode ssl.SSLContext*), 1214
set_quit() (*méthode bdb.Bdb*), 1872
set_result() (*méthode asyncio.Future*), 1132
set_result() (*méthode concurrent.futures.Future*), 1026
set_return() (*méthode bdb.Bdb*), 1872
set_running_or_notify_cancel() (*méthode concurrent.futures.Future*), 1026
set_selection() (*méthode tkinter.filedialog.FileDialog*), 1636
set_seq1() (*méthode difflib.SequenceMatcher*), 161
set_seq2() (*méthode difflib.SequenceMatcher*), 161
set_seqs() (*méthode difflib.SequenceMatcher*), 160
set_sequences() (*méthode mailbox.MH*), 1324
set_sequences() (*méthode mailbox.MHMessage*), 1331
set_server_documentation() (*méthode xmlrpc.server.DocCGIXMLRPCRequestHandler*), 1534
set_server_documentation() (*méthode xmlrpc.server.DocXMLRPCServer*), 1534
set_server_name() (*méthode xmlrpc.server.DocCGIXMLRPCRequestHandler*), 1534
set_server_name() (*méthode xmlrpc.server.DocXMLRPCServer*), 1534
set_server_title() (*méthode xmlrpc.server.DocCGIXMLRPCRequestHandler*), 1534
set_server_title() (*méthode xmlrpc.server.DocXMLRPCServer*), 1533
set_servername_callback (*attribut ssl.SSLContext*), 1210
set_start_method() (*dans le module multiprocessing*), 984
set_startup_hook() (*dans le module readline*), 177
set_step() (*méthode bdb.Bdb*), 1872
set_subdir() (*méthode mailbox.MaildirMessage*), 1327
set_tabsize() (*dans le module curses*), 929
set_task_factory() (*méthode asyncio.loop*), 1112
set_threshold() (*dans le module gc*), 2022
set_trace() (*dans le module bdb*), 1874
set_trace() (*dans le module pdb*), 1878
set_trace() (*méthode bdb.Bdb*), 1872
set_trace() (*méthode pdb.Pdb*), 1879
set_trace_callback() (*méthode sqlite3.Connection*), 547
set_tunnel() (*méthode http.client.HTTPConnection*), 1461
set_type() (*méthode email.message.Message*), 1294
set_unittest_reportflags() (*dans le module doctest*), 1742
set_unixfrom() (*méthode*

email.message.EmailMessage), 1252
set_unixfrom() (méthode email.message.Message), 1289
set_until() (méthode bdb.Bdb), 1872
SET_UPDATE (opcode), 2172
set_url() (méthode urllib.robotparser.RobotFileParser), 1453
set_usage() (méthode optparse.OptionParser), 913
set_userptr() (méthode curses.panel.Panel), 954
set_visible() (méthode mailbox.BabylMessage), 1332
set_wakeup_fd() (dans le module signal), 1240
set_write_buffer_limits() (méthode asyncio.WriteTransport), 1137
setacl() (méthode imaplib.IMAP4), 1479
setannotation() (méthode imaplib.IMAP4), 1479
setattr()
built-in function, 29
setAttribute() (méthode xml.dom.Element), 1380
setAttributeNode() (méthode xml.dom.Element), 1380
setAttributeNodeNS() (méthode xml.dom.Element), 1380
setAttributeNS() (méthode xml.dom.Element), 1381
SetBase() (méthode xml.parsers.expat.xmlparser), 1404
setblocking() (méthode socket.socket), 1184
setByteStream() (méthode xml.sax.xmlreader.InputSource), 1402
setcbreak() (dans le module tty), 2202
setCharacterStream() (méthode xml.sax.xmlreader.InputSource), 1402
SetComp (classe dans ast), 2113
setcomptype() (méthode wave.Wave_write), 1551
setconfig() (méthode sqlite3.Connection), 550
setContentHandler() (méthode xml.sax.xmlreader.XMLReader), 1400
setcontext() (dans le module decimal), 363
setDaemon() (méthode threading.Thread), 961
setdefault() (méthode dict), 93
setdefault() (méthode http.cookies.Morsel), 1511
setdefaulttimeout() (dans le module socket), 1177
setdlopenflags() (dans le module sys), 1948
setDocumentLocator() (méthode xml.sax.handler.ContentHandler), 1394
setDTDHandler() (méthode xml.sax.xmlreader.XMLReader), 1400
setegid() (dans le module os), 669
setEncoding() (méthode xml.sax.xmlreader.InputSource), 1401
setEntityResolver() (méthode xml.sax.xmlreader.XMLReader), 1400
setErrorHandler() (méthode xml.sax.xmlreader.XMLReader), 1400
seteuid() (dans le module os), 669
setFeature() (méthode xml.sax.xmlreader.XMLReader), 1400
setfirstweekday() (dans le module calendar), 254
setfirstweekday() (méthode calendar.Calendar), 250
setFormatter() (méthode logging.Handler), 763
setframerate() (méthode wave.Wave_write), 1551
setgid() (dans le module os), 670
setgroups() (dans le module os), 670
seth() (dans le module turtle), 1580
setheading() (dans le module turtle), 1580
sethostname() (dans le module socket), 1177
setinputsizes() (méthode sqlite3.Cursor), 554
setitem() (dans le module operator), 440
setitimer() (dans le module signal), 1239
setLevel() (méthode logging.Handler), 763
setLevel() (méthode logging.Logger), 758
setlimit() (méthode sqlite3.Connection), 550
setlocale() (dans le module locale), 1561
setLocale() (méthode xml.sax.xmlreader.XMLReader), 1400
setLoggerClass() (dans le module logging), 773
setlogmask() (dans le module syslog), 2211
setLogRecordFactory() (dans le module logging), 773
setMaxConns() (méthode urllib.request.CacheFTPHandler), 1438
setmode() (dans le module msvcrt), 2184
setName() (méthode threading.Thread), 960
setnchannels() (méthode wave.Wave_write), 1551
setnframes() (méthode wave.Wave_write), 1551
setns() (dans le module os), 670
setoutputsize() (méthode sqlite3.Cursor), 554
SetParamEntityParsing() (méthode xml.parsers.expat.xmlparser), 1404
setparams() (méthode wave.Wave_write), 1551
setpassword() (méthode zipfile.ZipFile), 589
setpgid() (dans le module os), 670
setpgrp() (dans le module os), 670
setpos() (dans le module turtle), 1579
setpos() (méthode wave.Wave_read), 1550
setposition() (dans le module turtle), 1579
setpriority() (dans le module os), 671
setprofile() (dans le module sys), 1948
setprofile() (dans le module threading), 957
setprofile_all_threads() (dans le module threading), 957
setProperty() (méthode xml.sax.xmlreader.XMLReader), 1401
setPublicId() (méthode xml.sax.xmlreader.InputSource), 1401
setquota() (méthode imaplib.IMAP4), 1479
setraw() (dans le module tty), 2202
setrecursionlimit() (dans le module sys), 1949
setregid() (dans le module os), 671
SetReparseDeferralEnabled() (méthode xml.parsers.expat.xmlparser), 1405
setresgid() (dans le module os), 671
setresuid() (dans le module os), 671
setreuid() (dans le module os), 671
setrlimit() (dans le module resource), 2207

setsampwidth() (*méthode wave.Wave_write*), 1551
setscrreg() (*méthode curses.window*), 936
setsid() (*dans le module os*), 671
setsockopt() (*méthode socket.socket*), 1184
setstate() (*dans le module random*), 384
setstate() (*méthode codecs.IncrementalDecoder*), 196
setstate() (*méthode codecs.IncrementalEncoder*), 195
setstate() (*méthode random.Random*), 388
setStream() (*méthode logging.StreamHandler*), 786
setswitchinterval() (*dans le module sys*), 1949
setswitchinterval() (*dans le module test.support*), 1850
setSystemId() (*méthode xml.sax.xmlreader.InputSource*), 1401
setsyx() (*dans le module curses*), 929
setTarget() (*méthode logging.handlers.MemoryHandler*), 797
settimeout() (*méthode socket.socket*), 1184
setTimeout() (*méthode urllib.request.CacheFTPHandler*), 1438
settrace() (*dans le module sys*), 1949
settrace() (*dans le module threading*), 957
settrace_all_threads() (*dans le module threading*), 957
setuid() (*dans le module os*), 671
setundobuffer() (*dans le module turtle*), 1594
--setup
    option de ligne de commande timeit, 1896
setup() (*dans le module turtle*), 1602
setup() (*méthode socketserver.BaseRequestHandler*), 1498
setUp() (*méthode unittest.TestCase*), 1761
SETUP_ANNOTATIONS (*opcode*), 2169
SETUP_CLEANUP (*opcode*), 2180
setup_environ() (*méthode wsgiref.handlers.BaseHandler*), 1423
SETUP_FINALLY (*opcode*), 2180
setup_python() (*méthode venv.EnvBuilder*), 1919
setup_scripts() (*méthode venv.EnvBuilder*), 1919
setup_testing_defaults() (*dans le module wsgiref.util*), 1417
SETUP_WITH (*opcode*), 2180
setUpClass() (*méthode unittest.TestCase*), 1761
setupterm() (*dans le module curses*), 929
SetValue() (*dans le module winreg*), 2190
SetValueEx() (*dans le module winreg*), 2190
setworldcoordinates() (*dans le module turtle*), 1596
setx() (*dans le module turtle*), 1580
setxattr() (*dans le module os*), 711
sety() (*dans le module turtle*), 1580
SF_APPEND (*dans le module stat*), 483
SF_ARCHIVED (*dans le module stat*), 483
SF_DATALESS (*dans le module stat*), 484
SF_FIRMLINK (*dans le module stat*), 484
SF_IMMUTABLE (*dans le module stat*), 483
SF_MNOWAIT (*dans le module os*), 683
SF_NOCACHE (*dans le module os*), 683
SF_NODISKIO (*dans le module os*), 683
SF_NOUNLINK (*dans le module stat*), 484
SF_RESTRICTED (*dans le module stat*), 483
SF_SETTABLE (*dans le module stat*), 483
SF_SNAPSHOT (*dans le module stat*), 484
SF_SUPPORTED (*dans le module stat*), 483
SF_SYNC (*dans le module os*), 683
SF_SYNTHETIC (*dans le module stat*), 483
sha1() (*dans le module hashlib*), 646
sha3_224() (*dans le module hashlib*), 647
sha3_256() (*dans le module hashlib*), 647
sha3_384() (*dans le module hashlib*), 647
sha3_512() (*dans le module hashlib*), 647
sha224() (*dans le module hashlib*), 647
sha256() (*dans le module hashlib*), 647
sha384() (*dans le module hashlib*), 647
sha512() (*dans le module hashlib*), 647
shake_128() (*dans le module hashlib*), 648
shake_256() (*dans le module hashlib*), 648
shape (*attribut memoryview*), 87
Shape (*classe dans turtle*), 1603
shape() (*dans le module turtle*), 1590
shapesize() (*dans le module turtle*), 1590
shapetransform() (*dans le module turtle*), 1591
share() (*méthode socket.socket*), 1185
ShareableList (*classe dans multiprocessing.shared_memory*), 1017
ShareableList() (*méthode multiprocessing.managers.SharedMemoryManager*), 1017
Shared Memory, 1014
shared_ciphers() (*méthode ssl.SSLSocket*), 1204
shared_memory (*attribut sys._emscripten_info*), 1933
SharedMemory (*classe dans multiprocessing.shared_memory*), 1014
SharedMemory() (*méthode multiprocessing.managers.SharedMemoryManager*), 1017
SharedMemoryManager (*classe dans multiprocessing.managers*), 1016
shearfactor() (*dans le module turtle*), 1591
Shelf (*classe dans shelve*), 527
shelve
    module, 526, 529
shield() (*dans le module asyncio*), 1073
shift() (*méthode decimal.Context*), 369
shift() (*méthode decimal.Decimal*), 362
shift_path_info() (*dans le module wsgiref.util*), 1416
shifting
    operations, 40
shlex
    module, 1613
shlex (*classe dans shlex*), 1614

shm (*attribut multiprocessing.shared_memory.ShareableList*), 1018
SHORT_TIMEOUT (*dans le module test.support*), 1848
shortDescription() (*méthode unittest.TestCase*), 1770
shorten() (*dans le module textwrap*), 169
shouldFlush() (*méthode logging.handlers.BufferingHandler*), 796
shouldFlush() (*méthode logging.handlers.MemoryHandler*), 797
shouldStop (*attribut unittest.TestResult*), 1777
show() (*méthode curses.panel.Panel*), 954
show() (*méthode tkinter.commondialog.Dialog*), 1637
show() (*méthode tkinter.messagebox.Message*), 1638
show_code() (*dans le module dis*), 2162
show_flag_values() (*dans le module enum*), 330
--show-caches
    option de ligne de commande dis, 2160
showerror() (*dans le module tkinter.messagebox*), 1638
showinfo() (*dans le module tkinter.messagebox*), 1638
--show-offsets
    option de ligne de commande dis, 2161
--show-positions
    option de ligne de commande dis, 2161
showsyntaxerror() (*méthode code.InteractiveInterpreter*), 2054
showtraceback() (*méthode code.InteractiveInterpreter*), 2054
showturtle() (*dans le module turtle*), 1589
showwarning() (*dans le module tkinter.messagebox*), 1638
showwarning() (*dans le module warnings*), 1977
shuffle() (*dans le module random*), 386
SHUT_RD (*dans le module socket*), 1171
SHUT_RDWR (*dans le module socket*), 1171
SHUT_WR (*dans le module socket*), 1171
ShutDown, 1050
shutdown() (*dans le module logging*), 772
shutdown() (*méthode asyncio.Queue*), 1103
shutdown() (*méthode concurrent.futures.Executor*), 1020
shutdown() (*méthode imaplib.IMAP4*), 1479
shutdown() (*méthode multiprocessing.managers.BaseManager*), 992
shutdown() (*méthode queue.Queue*), 1051
shutdown() (*méthode socketserver.BaseServer*), 1496
shutdown() (*méthode socket.socket*), 1185
shutdown_asyncgens() (*méthode asyncio.loop*), 1109
shutdown_default_executor() (*méthode asyncio.loop*), 1110
shutil
    module, 497
SI (*dans le module curses.ascii*), 950
side_effect (*attribut unittest.mock.Mock*), 1791
SIG_BLOCK (*dans le module signal*), 1237
SIG_DFL (*dans le module signal*), 1235
SIG_IGN (*dans le module signal*), 1235
SIG_SETMASK (*dans le module signal*), 1238
SIG_UNBLOCK (*dans le module signal*), 1238
SIGABRT (*dans le module signal*), 1235
SIGALRM (*dans le module signal*), 1236
SIGBREAK (*dans le module signal*), 1236
SIGBUS (*dans le module signal*), 1236
SIGCHLD (*dans le module signal*), 1236
SIGCLD (*dans le module signal*), 1236
SIGCONT (*dans le module signal*), 1236
SIGFPE (*dans le module signal*), 1236
SIGHUP (*dans le module signal*), 1236
SIGILL (*dans le module signal*), 1236
SIGINT (*dans le module signal*), 1236
siginterrupt() (*dans le module signal*), 1240
SIGKILL (*dans le module signal*), 1236
Sigmasks (*classe dans signal*), 1235
signal
    module, 1059, 1234
signal() (*dans le module signal*), 1240
Signals (*classe dans signal*), 1235
signature (*attribut inspect.BoundArguments*), 2036
Signature (*classe dans inspect*), 2032
signature() (*dans le module inspect*), 2031
sigpending() (*dans le module signal*), 1241
SIGPIPE (*dans le module signal*), 1236
SIGSEGV (*dans le module signal*), 1236
SIGSTKFLT (*dans le module signal*), 1237
SIGTERM (*dans le module signal*), 1237
sigtimedwait() (*dans le module signal*), 1241
SIGUSR1 (*dans le module signal*), 1237
SIGUSR2 (*dans le module signal*), 1237
sigwait() (*dans le module signal*), 1241
sigwaitinfo() (*dans le module signal*), 1241
SIGWINCH (*dans le module signal*), 1237
SIMPLE (*attribut inspect.BufferFlags*), 2044
Simple Mail Transfer Protocol, 1482
SimpleCookie (*classe dans http.cookies*), 1509
simplefilter() (*dans le module warnings*), 1977
SimpleHandler (*classe dans wsgiref.handlers*), 1421
SimpleHTTPRequestHandler (*classe dans http.server*), 1506
SimpleNamespace (*classe dans types*), 304
SimpleQueue (*classe dans multiprocessing*), 981
SimpleQueue (*classe dans queue*), 1049
SimpleXMLRPCRequestHandler (*classe dans xmlrpc.server*), 1529
SimpleXMLRPCServer (*classe dans xmlrpc.server*), 1529
sin() (*dans le module cmath*), 349
sin() (*dans le module math*), 345
SingleAddressHeader (*classe dans email.headerregistry*), 1275
singledispatch() (*dans le module functools*), 432
singledispatchmethod (*classe dans functools*), 435
sinh() (*dans le module cmath*), 349
sinh() (*dans le module math*), 346
SIO_KEEPALIVE_VALS (*dans le module socket*), 1169

SIO_LOOPBACK_FAST_PATH (*dans le module socket*), 1169
SIO_RCVALL (*dans le module socket*), 1169
site
    module, 2049
site_maps() (*méthode urllib.robotparser.RobotFileParser*), 1453
sitecustomize
    module, 2050
site-packages
    directory, 2049
sixtofour (*attribut ipaddress.IPv6Address*), 1538
size (*attribut multiprocessing.shared_memory.SharedMemory*), 1015
size (*attribut struct.Struct*), 188
size (*attribut tarfile.TarInfo*), 603
size (*attribut tracemalloc.Statistic*), 1909
size (*attribut tracemalloc.StatisticDiff*), 1909
size (*attribut tracemalloc.Trace*), 1910
size() (*méthode ftplib.FTP*), 1469
size() (*méthode mmap.mmap*), 1247
size_diff (*attribut tracemalloc.StatisticDiff*), 1909
Sized (*classe dans collections.abc*), 279
Sized (*classe dans typing*), 1723
sizeof() (*dans le module ctypes*), 841
sizeof_digit (*attribut sys.int_info*), 1944
SKIP (*dans le module doctest*), 1736
skip() (*dans le module unittest*), 1759
skip_if_broken_multiprocessing_synchronize() (*dans le module test.support*), 1855
skip_unless_bind_unix_socket() (*dans le module test.support.socket_helper*), 1856
skip_unless_symlink() (*dans le module test.support.os_helper*), 1860
skip_unless_xattr() (*dans le module test.support.os_helper*), 1860
skipIf() (*dans le module unittest*), 1759
skipinitialspace (*attribut csv.Dialect*), 618
skipped (*attribut doctest.TestResults*), 1746
skipped (*attribut unittest.TestResult*), 1776
skippedEntity() (*méthode xml.sax.handler.ContentHandler*), 1396
skips (*attribut doctest.DocTestRunner*), 1747
SkipTest, 1760
skipTest() (*méthode unittest.TestCase*), 1762
skipUnless() (*dans le module unittest*), 1759
SLASH (*dans le module token*), 2144
SLASHEQUAL (*dans le module token*), 2145
sleep() (*dans le module asyncio*), 1071
sleep() (*dans le module time*), 747
sleeping_retry() (*dans le module test.support*), 1849
slice
    assignment, 49
    built-in function, 2177
    operation, 47
Slice (*classe dans ast*), 2113
slice (*classe de base*), 29
slow_callback_duration (*attribut asyncio.loop*), 1124
SMALLEST (*dans le module test.support*), 1849
SMTP
    protocol, 1482
SMTP (*classe dans smtplib*), 1482
SMTP (*dans le module email.policy*), 1270
SMTP_SSL (*classe dans smtplib*), 1482
SMTPAuthenticationError, 1483
SMTPConnectError, 1483
smtpd
    module, 2224
SMTPDataError, 1483
SMTPException, 1483
SMTPHandler (*classe dans logging.handlers*), 796
SMTPHeloError, 1483
smtplib
    module, 1482
SMTPNotSupportedError, 1483
SMTPRecipientsRefused, 1483
SMTPResponseException, 1483
SMTPSenderRefused, 1483
SMTPServerDisconnected, 1483
SMTPUTF8 (*dans le module email.policy*), 1270
Snapshot (*classe dans tracemalloc*), 1908
SND_ALIAS (*dans le module winsound*), 2194
SND_ASYNC (*dans le module winsound*), 2195
SND_FILENAME (*dans le module winsound*), 2194
SND_LOOP (*dans le module winsound*), 2195
SND_MEMORY (*dans le module winsound*), 2195
SND_NODEFAULT (*dans le module winsound*), 2195
SND_NOSTOP (*dans le module winsound*), 2195
SND_NOWAIT (*dans le module winsound*), 2195
SND_PURGE (*dans le module winsound*), 2195
sndhdr
    module, 2224
sni_callback (*attribut ssl.SSLContext*), 1209
sniff() (*méthode csv.Sniffer*), 616
Sniffer (*classe dans csv*), 616
SO (*dans le module curses.ascii*), 950
SO_INCOMING_CPU (*dans le module socket*), 1170
SO_REUSEPORT_LB (*dans le module socket*), 1170
sock_accept() (*méthode asyncio.loop*), 1119
SOCK_CLOEXEC (*dans le module socket*), 1166
sock_connect() (*méthode asyncio.loop*), 1119
SOCK_DGRAM (*dans le module socket*), 1166
SOCK_MAX_SIZE (*dans le module test.support*), 1848
SOCK_NONBLOCK (*dans le module socket*), 1166
SOCK_RAW (*dans le module socket*), 1166
SOCK_RDM (*dans le module socket*), 1166
sock_recv() (*méthode asyncio.loop*), 1118
sock_recv_into() (*méthode asyncio.loop*), 1118
sock_recvfrom() (*méthode asyncio.loop*), 1119
sock_recvfrom_into() (*méthode asyncio.loop*), 1119
sock_sendall() (*méthode asyncio.loop*), 1119
sock_sendfile() (*méthode asyncio.loop*), 1120
sock_sendto() (*méthode asyncio.loop*), 1119

SOCK_SEQPACKET (*dans le module socket*), 1166
SOCK_STREAM (*dans le module socket*), 1166
socket
    module, 1162, 1413
    object, 1162
socket (*attribut socketserver.BaseServer*), 1497
socket (*classe dans socket*), 1171
socket() (*in module socket*), 1225
socket() (*méthode imaplib.IMAP4*), 1480
socket_type (*attribut socketserver.BaseServer*), 1497
SocketHandler (*classe dans logging.handlers*), 791
socketpair() (*dans le module socket*), 1171
sockets (*attribut asyncio.Server*), 1127
socketserver
    module, 1493
SocketType (*dans le module socket*), 1173
soft deprecated, **2243**
SOFT_KEYWORD (*dans le module token*), 2146
softkwlist (*dans le module keyword*), 2147
SOH (*dans le module curses.ascii*), 950
SOL_ALG (*dans le module socket*), 1169
SOL_RDS (*dans le module socket*), 1169
SOMAXCONN (*dans le module socket*), 1166
sort() (*méthode imaplib.IMAP4*), 1480
sort() (*méthode list*), 50
sort_stats() (*méthode pstats.Stats*), 1890
sortdict() (*dans le module test.support*), 1850
sorted()
    built-in function, 29
--sort-keys
    option de ligne de commande json, 1316
sortTestMethodsUsing (*attribut unittest.TestLoader*), 1776
source (*attribut doctest.Example*), 1744
source (*attribut shlex.shlex*), 1616
source (*pdb command*), 1882
SOURCE_DATE_EPOCH, 2154, 2155, 2157
source_from_cache() (*dans le module importlib.util*), 2080
source_hash() (*dans le module importlib.util*), 2081
SOURCE_SUFFIXES (*dans le module importlib.machinery*), 2074
source_to_code() (*méthode statique importlib.abc.InspectLoader*), 2070
SourceFileLoader (*classe dans importlib.machinery*), 2076
sourcehook() (*méthode shlex.shlex*), 1615
SourcelessFileLoader (*classe dans importlib.machinery*), 2077
SourceLoader (*classe dans importlib.abc*), 2071
SP (*dans le module curses.ascii*), 951
space
    in printf-style formatting, 63, 79
--spacing
    option de ligne de commande calendar, 258
span() (*méthode re.Match*), 150
sparse (*attribut tarfile.TarInfo*), 604
spawn() (*dans le module pty*), 2202
spawn_python() (*dans le module test.support.script_helper*), 1857
spawnl() (*dans le module os*), 719
spawnle() (*dans le module os*), 719
spawnlp() (*dans le module os*), 719
spawnlpe() (*dans le module os*), 719
spawnv() (*dans le module os*), 719
spawnve() (*dans le module os*), 719
spawnvp() (*dans le module os*), 719
spawnvpe() (*dans le module os*), 719
spec_from_file_location() (*dans le module importlib.util*), 2081
spec_from_loader() (*dans le module importlib.util*), 2081
special
    méthode, 2243
SpecialFileError, 598
--specialized
    option de ligne de commande dis, 2161
spécificateur de module, **2239**
specified_attributes (*attribut xml.parsers.expat.xmlparser*), 1405
speed() (*dans le module turtle*), 1583
Spinbox (*classe dans tkinter.ttk*), 1646
splice() (*dans le module os*), 683
SPLICE_F_MORE (*dans le module os*), 683
SPLICE_F_MOVE (*dans le module os*), 683
SPLICE_F_NONBLOCK (*dans le module os*), 683
split() (*dans le module os.path*), 477
split() (*dans le module re*), 143
split() (*dans le module shlex*), 1613
split() (*méthode BaseExceptionGroup*), 118
split() (*méthode bytearray*), 73
split() (*méthode bytes*), 73
split() (*méthode re.Pattern*), 147
split() (*méthode str*), 60
splitdrive() (*dans le module os.path*), 477
splitext() (*dans le module os.path*), 478
splitlines() (*méthode bytearray*), 76
splitlines() (*méthode bytes*), 76
splitlines() (*méthode str*), 60
SplitResult (*classe dans urllib.parse*), 1449
SplitResultBytes (*classe dans urllib.parse*), 1449
splitroot() (*dans le module os.path*), 478
SpooledTemporaryFile (*classe dans tempfile*), 488
sprintf-style formatting, 63, 79
spwd
    module, 2224
sqlite3
    module, 536
SQLITE_DBCONFIG_DEFENSIVE (*dans le module sqlite3*), 542
SQLITE_DBCONFIG_DQS_DDL (*dans le module sqlite3*), 542
SQLITE_DBCONFIG_DQS_DML (*dans le module sqlite3*), 542

SQLITE_DBCONFIG_ENABLE_FKEY (*dans le module sqlite3*), 542
SQLITE_DBCONFIG_ENABLE_FTS3_TOKENIZER (*dans le module sqlite3*), 542
SQLITE_DBCONFIG_ENABLE_LOAD_EXTENSION (*dans le module sqlite3*), 542
SQLITE_DBCONFIG_ENABLE_QPSG (*dans le module sqlite3*), 542
SQLITE_DBCONFIG_ENABLE_TRIGGER (*dans le module sqlite3*), 542
SQLITE_DBCONFIG_ENABLE_VIEW (*dans le module sqlite3*), 542
SQLITE_DBCONFIG_LEGACY_ALTER_TABLE (*dans le module sqlite3*), 542
SQLITE_DBCONFIG_LEGACY_FILE_FORMAT (*dans le module sqlite3*), 542
SQLITE_DBCONFIG_NO_CKPT_ON_CLOSE (*dans le module sqlite3*), 542
SQLITE_DBCONFIG_RESET_DATABASE (*dans le module sqlite3*), 542
SQLITE_DBCONFIG_TRIGGER_EQP (*dans le module sqlite3*), 542
SQLITE_DBCONFIG_TRUSTED_SCHEMA (*dans le module sqlite3*), 542
SQLITE_DBCONFIG_WRITABLE_SCHEMA (*dans le module sqlite3*), 542
SQLITE_DENY (*dans le module sqlite3*), 541
sqlite_errorcode (*attribut sqlite3.Error*), 556
sqlite_errorname (*attribut sqlite3.Error*), 556
SQLITE_IGNORE (*dans le module sqlite3*), 541
SQLITE_OK (*dans le module sqlite3*), 541
sqlite_version (*dans le module sqlite3*), 541
sqlite_version_info (*dans le module sqlite3*), 541
sqrt() (*dans le module cmath*), 348
sqrt() (*dans le module math*), 344
sqrt() (*méthode decimal.Context*), 369
sqrt() (*méthode decimal.Decimal*), 362
SSL, 1189
ssl
    module, 1189
ssl_version (*attribut ftplib.FTP_TLS*), 1470
SSLCertVerificationError, 1193
SSLContext (*classe dans ssl*), 1206
SSLEOFError, 1193
SSLError, 1192
SSLErrorNumber (*classe dans ssl*), 1201
SSLKEYLOGFILE, 1191, 1192
SSLObject (*classe dans ssl*), 1220
sslobject_class (*attribut ssl.SSLContext*), 1211
SSLSession (*classe dans ssl*), 1222
SSLSocket (*classe dans ssl*), 1202
sslsocket_class (*attribut ssl.SSLContext*), 1211
SSLSyscallError, 1193
SSLv3 (*attribut ssl.TLSVersion*), 1201
SSLWantReadError, 1193
SSLWantWriteError, 1193
SSLZeroReturnError, 1192
st() (*dans le module turtle*), 1589
st_atime (*attribut os.stat_result*), 699
ST_ATIME (*dans le module stat*), 480
st_atime_ns (*attribut os.stat_result*), 699
st_birthtime (*attribut os.stat_result*), 700
st_birthtime_ns (*attribut os.stat_result*), 700
st_blksize (*attribut os.stat_result*), 700
st_blocks (*attribut os.stat_result*), 700
st_creator (*attribut os.stat_result*), 700
st_ctime (*attribut os.stat_result*), 699
ST_CTIME (*dans le module stat*), 481
st_ctime_ns (*attribut os.stat_result*), 699
st_dev (*attribut os.stat_result*), 699
ST_DEV (*dans le module stat*), 480
st_file_attributes (*attribut os.stat_result*), 701
st_flags (*attribut os.stat_result*), 700
st_fstype (*attribut os.stat_result*), 700
st_gen (*attribut os.stat_result*), 700
st_gid (*attribut os.stat_result*), 699
ST_GID (*dans le module stat*), 480
st_ino (*attribut os.stat_result*), 699
ST_INO (*dans le module stat*), 480
st_mode (*attribut os.stat_result*), 699
ST_MODE (*dans le module stat*), 480
st_mtime (*attribut os.stat_result*), 699
ST_MTIME (*dans le module stat*), 480
st_mtime_ns (*attribut os.stat_result*), 699
st_nlink (*attribut os.stat_result*), 699
ST_NLINK (*dans le module stat*), 480
st_rdev (*attribut os.stat_result*), 700
st_reparse_tag (*attribut os.stat_result*), 701
st_rsize (*attribut os.stat_result*), 700
st_size (*attribut os.stat_result*), 699
ST_SIZE (*dans le module stat*), 480
st_type (*attribut os.stat_result*), 700
st_uid (*attribut os.stat_result*), 699
ST_UID (*dans le module stat*), 480
stack (*attribut traceback.TracebackException*), 2014
stack viewer, 1662
stack() (*dans le module inspect*), 2040
stack_effect() (*dans le module dis*), 2164
stack_size() (*dans le module _thread*), 1058
stack_size() (*dans le module threading*), 957
stackable
    streams, 189
StackSummary (*classe dans traceback*), 2015
stamp() (*dans le module turtle*), 1582
standard_b64decode() (*dans le module base64*), 1340
standard_b64encode() (*dans le module base64*), 1340
standend() (*méthode curses.window*), 936
standout() (*méthode curses.window*), 936
STAR (*dans le module token*), 2144
STAREQUAL (*dans le module token*), 2145
starmap() (*dans le module itertools*), 418
starmap() (*méthode multiprocessing.pool.Pool*), 999
starmap_async() (*méthode multiprocessing.pool.Pool*), 999

Starred (*classe dans ast*), 2109
start (*attribut range*), 51
start (*attribut slice*), 29
start (*attribut UnicodeError*), 114
start() (*dans le module tracemalloc*), 1906
start() (*méthode logging.handlers.QueueListener*), 799
start() (*méthode multiprocessing.managers.BaseManager*), 991
start() (*méthode multiprocessing.Process*), 977
start() (*méthode re.Match*), 149
start() (*méthode threading.Thread*), 959
start() (*méthode tkinter.ttk.Progressbar*), 1648
start() (*méthode xml.etree.ElementTree.TreeBuilder*), 1371
start_color() (*dans le module curses*), 929
start_new_thread() (*dans le module _thread*), 1057
start_ns() (*méthode xml.etree.ElementTree.TreeBuilder*), 1371
start_server() (*dans le module asyncio*), 1085
start_serving() (*méthode asyncio.Server*), 1127
start_threads() (*dans le module test.support.threading_helper*), 1858
start_tls() (*méthode asyncio.loop*), 1117
start_tls() (*méthode asyncio.StreamWriter*), 1088
start_unix_server() (*dans le module asyncio*), 1086
startCDATA() (*méthode xml.sax.handler.LexicalHandler*), 1397
StartCdataSectionHandler() (*méthode xml.parsers.expat.xmlparser*), 1407
--start-directory
    option de ligne de commande unittest-discover, 1755
StartDoctypeDeclHandler() (*méthode xml.parsers.expat.xmlparser*), 1406
startDocument() (*méthode xml.sax.handler.ContentHandler*), 1395
startDTD() (*méthode xml.sax.handler.LexicalHandler*), 1397
startElement() (*méthode xml.sax.handler.ContentHandler*), 1395
StartElementHandler() (*méthode xml.parsers.expat.xmlparser*), 1407
startElementNS() (*méthode xml.sax.handler.ContentHandler*), 1395
STARTF_FORCEOFFFEEDBACK (*dans le module subprocess*), 1041
STARTF_FORCEONFEEDBACK (*dans le module subprocess*), 1041
STARTF_USESHOWWINDOW (*dans le module subprocess*), 1041
STARTF_USESTDHANDLES (*dans le module subprocess*), 1040
startfile() (*dans le module os*), 720
StartNamespaceDeclHandler() (*méthode xml.parsers.expat.xmlparser*), 1407
startPrefixMapping() (*méthode xml.sax.handler.ContentHandler*), 1395
StartResponse (*classe dans wsgiref.types*), 1424
startswith() (*méthode bytearray*), 71
startswith() (*méthode bytes*), 71
startswith() (*méthode str*), 61
startTest() (*méthode unittest.TestResult*), 1777
startTestRun() (*méthode unittest.TestResult*), 1778
starttls() (*méthode imaplib.IMAP4*), 1480
starttls() (*méthode smtplib.SMTP*), 1486
STARTUPINFO (*classe dans subprocess*), 1039
stat
    module, 479, 698
stat() (*dans le module os*), 698
stat() (*méthode os.DirEntry*), 697
stat() (*méthode pathlib.Path*), 460
stat() (*méthode poplib.POP3*), 1472
stat_result (*classe dans os*), 699
state() (*méthode tkinter.ttk.Widget*), 1644
statement
    assert, 109
    del, 49, 90
    except, 107
    if, 37
    import, 33, 2049
    raise, 107
    try, 107
    while, 37
static type checker, **2243**
static_order() (*méthode graphlib.TopologicalSorter*), 332
staticmethod()
    built-in function, 30
Statistic (*classe dans tracemalloc*), 1909
StatisticDiff (*classe dans tracemalloc*), 1909
statistics
    module, 393
statistics() (*méthode tracemalloc.Snapshot*), 1908
StatisticsError, 404
Stats (*classe dans pstats*), 1889
status (*attribut http.client.HTTPResponse*), 1463
status (*attribut urllib.response.addinfourl*), 1442
status() (*méthode imaplib.IMAP4*), 1480
statvfs() (*dans le module os*), 701
STD_ERROR_HANDLE (*dans le module subprocess*), 1040
STD_INPUT_HANDLE (*dans le module subprocess*), 1040
STD_OUTPUT_HANDLE (*dans le module subprocess*), 1040
stderr (*attribut asyncio.subprocess.Process*), 1101
stderr (*attribut subprocess.CalledProcessError*), 1031
stderr (*attribut subprocess.CompletedProcess*), 1030
stderr (*attribut subprocess.Popen*), 1039
stderr (*attribut subprocess.TimeoutExpired*), 1031
stderr (*dans le module sys*), 1951
stdev (*attribut statistics.NormalDist*), 404
stdev() (*dans le module statistics*), 401
stdin (*attribut asyncio.subprocess.Process*), 1100
stdin (*attribut subprocess.Popen*), 1039
stdin (*dans le module sys*), 1951
stdlib_module_names (*dans le module sys*), 1952

stdout (*attribut asyncio.subprocess.Process*), 1101
stdout (*attribut subprocess.CalledProcessError*), 1031
stdout (*attribut subprocess.CompletedProcess*), 1030
stdout (*attribut subprocess.Popen*), 1039
stdout (*attribut subprocess.TimeoutExpired*), 1031
STDOUT (*dans le module subprocess*), 1030
stdout (*dans le module sys*), 1951
stem (*attribut pathlib.PurePath*), 453
step (*attribut range*), 51
step (*attribut slice*), 29
step (*pdb command*), 1881
step() (*méthode tkinter.ttk.Progressbar*), 1648
stls() (*méthode poplib.POP3*), 1473
stop (*attribut range*), 51
stop (*attribut slice*), 29
stop() (*dans le module tracemalloc*), 1907
stop() (*méthode asyncio.loop*), 1109
stop() (*méthode logging.handlers.QueueListener*), 799
stop() (*méthode tkinter.ttk.Progressbar*), 1648
stop() (*méthode unittest.TestResult*), 1777
stop_here() (*méthode bdb.Bdb*), 1871
STOP_ITERATION (*monitoring event*), 1956
StopAsyncIteration, 112
StopIteration, 112
stopListening() (*dans le module logging.config*), 776
stopTest() (*méthode unittest.TestResult*), 1777
stopTestRun() (*méthode unittest.TestResult*), 1778
storbinary() (*méthode ftplib.FTP*), 1467
Store (*classe dans ast*), 2109
store() (*méthode imaplib.IMAP4*), 1480
STORE_ACTIONS (*attribut optparse.Option*), 919
STORE_ATTR (*opcode*), 2171
STORE_DEREF (*opcode*), 2175
STORE_FAST (*opcode*), 2175
STORE_FAST_LOAD_FAST (*opcode*), 2175
STORE_FAST_STORE_FAST (*opcode*), 2175
STORE_GLOBAL (*opcode*), 2171
STORE_NAME (*opcode*), 2170
STORE_SLICE (*opcode*), 2168
STORE_SUBSCR (*opcode*), 2167
storlines() (*méthode ftplib.FTP*), 1467
str (*built-in class*)
    (see also string), 52
str (*classe de base*), 54
str() (*dans le module locale*), 1567
str_digits_check_threshold (*attribut sys.int_info*), 1944
strcoll() (*dans le module locale*), 1566
StreamError, 597
StreamHandler (*classe dans logging*), 786
streamreader (*attribut codecs.CodecInfo*), 189
StreamReader (*classe dans asyncio*), 1087
StreamReader (*classe dans codecs*), 196
StreamReaderWriter (*classe dans codecs*), 197
StreamRecoder (*classe dans codecs*), 198
StreamRequestHandler (*classe dans socketserver*), 1498
streams, 189
    stackable, 189
streamwriter (*attribut codecs.CodecInfo*), 189
StreamWriter (*classe dans asyncio*), 1088
StreamWriter (*classe dans codecs*), 196
StrEnum (*classe dans enum*), 323
strerror (*attribut OSError*), 111
strerror() (*dans le module os*), 671
strftime() (*dans le module time*), 748
strftime() (*méthode datetime.date*), 217
strftime() (*méthode datetime.datetime*), 227
strftime() (*méthode datetime.time*), 232
strict
    error handler's name, 192
strict (*attribut csv.Dialect*), 618
STRICT (*attribut enum.FlagBoundary*), 327
strict (*dans le module email.policy*), 1271
strict_domain (*attribut http.cookiejar.DefaultCookiePolicy*), 1518
strict_errors() (*dans le module codecs*), 193
strict_ns_domain (*attribut http.cookiejar.DefaultCookiePolicy*), 1518
strict_ns_set_initial_dollar (*attribut http.cookiejar.DefaultCookiePolicy*), 1518
strict_ns_set_path (*attribut http.cookiejar.DefaultCookiePolicy*), 1518
strict_ns_unverifiable (*attribut http.cookiejar.DefaultCookiePolicy*), 1518
strict_rfc2965_unverifiable (*attribut http.cookiejar.DefaultCookiePolicy*), 1518
STRIDED (*attribut inspect.BufferFlags*), 2044
STRIDED_RO (*attribut inspect.BufferFlags*), 2044
STRIDES (*attribut inspect.BufferFlags*), 2044
strides (*attribut memoryview*), 87
string
    format() (*built-in function*), 17
    formatting, printf, 63
    interpolation, printf, 63
    methods, 54
    module, 121
    object, 52
    str (*built-in class*), 54
    str() (*built-in function*), 30
    text sequence type, 52
STRING (*attribut annotationlib.Format*), 2046
string (*attribut re.Match*), 150
STRING (*dans le module token*), 2144
string_at() (*dans le module ctypes*), 841
StringIO (*classe dans io*), 742
stringprep
    module, 174
strip() (*méthode bytearray*), 73
strip() (*méthode bytes*), 73
strip() (*méthode str*), 61
strip_dirs() (*méthode pstats.Stats*), 1890
stripspaces (*attribut curses.textpad.Textbox*), 949
strptime() (*dans le module time*), 750
strptime() (*méthode de la classe datetime.date*), 214

strptime() (*méthode de la classe datetime.datetime*), 221
strptime() (*méthode de la classe datetime.time*), 231
strsignal() (*dans le module signal*), 1238
struct
    module, 181, 1185
Struct (*classe dans struct*), 188
struct_time (*classe dans time*), 750
Structure (*classe dans ctypes*), 846
structures
    C, 181
strxfrm() (*dans le module locale*), 1566
STX (*dans le module curses.ascii*), 950
Style (*classe dans tkinter.ttk*), 1655
Sub (*classe dans ast*), 2110
SUB (*dans le module curses.ascii*), 951
sub() (*dans le module operator*), 439
sub() (*dans le module re*), 144
sub() (*méthode re.Pattern*), 147
subdirs (*attribut filecmp.dircmp*), 486
SubElement() (*dans le module xml.etree.ElementTree*), 1364
subgroup() (*méthode BaseExceptionGroup*), 117
submit() (*méthode concurrent.futures.Executor*), 1020
submodule_search_locations (*attribut importlib.machinery.ModuleSpec*), 2078
subn() (*dans le module re*), 145
subn() (*méthode re.Pattern*), 147
subnet_of() (*méthode ipaddress.IPv4Network*), 1542
subnet_of() (*méthode ipaddress.IPv6Network*), 1544
subnets() (*méthode ipaddress.IPv4Network*), 1542
subnets() (*méthode ipaddress.IPv6Network*), 1544
Subnormal (*classe dans decimal*), 371
suboffsets (*attribut memoryview*), 88
subpad() (*méthode curses.window*), 936
subprocess
    module, 1028
subprocess_exec() (*méthode asyncio.loop*), 1124
subprocess_shell() (*méthode asyncio.loop*), 1125
SubprocessError, 1030
SubprocessProtocol (*classe dans asyncio*), 1139
SubprocessTransport (*classe dans asyncio*), 1135
subscribe() (*méthode imaplib.IMAP4*), 1480
subscript
    assignment, 49
    operation, 47
Subscript (*classe dans ast*), 2112
subsequent_indent (*attribut textwrap.TextWrapper*), 171
substitute() (*méthode string.Template*), 130
subTest() (*méthode unittest.TestCase*), 1762
subtract() (*méthode collections.Counter*), 262
subtract() (*méthode decimal.Context*), 369
subtype (*attribut email.headerregistry.ContentTypeHeader*), 1276
subwin() (*méthode curses.window*), 936
successful() (*méthode multiprocessing.pool.AsyncResult*), 1000
suffix (*attribut pathlib.PurePath*), 452
suffix_map (*attribut mimetypes.MimeTypes*), 1338
suffix_map (*dans le module mimetypes*), 1337
suffixes (*attribut pathlib.PurePath*), 452
suiteClass (*attribut unittest.TestLoader*), 1776
sum()
    built-in function, 30
summarize() (*méthode doctest.DocTestRunner*), 1747
summarize_address_range() (*dans le module ipaddress*), 1547
--summary
    option de ligne de commande trace, 1899
sumprod() (*dans le module math*), 345
sunau
    module, 2224
SUNDAY (*dans le module calendar*), 255
super (*attribut pyclbr.Class*), 2154
super (*classe de base*), 30
supernet() (*méthode ipaddress.IPv4Network*), 1542
supernet() (*méthode ipaddress.IPv6Network*), 1544
supernet_of() (*méthode ipaddress.IPv4Network*), 1542
supernet_of() (*méthode ipaddress.IPv6Network*), 1544
supports_bytes_environ (*dans le module os*), 671
supports_dir_fd (*dans le module os*), 702
supports_effective_ids (*dans le module os*), 702
supports_fd (*dans le module os*), 702
supports_follow_symlinks (*dans le module os*), 702
supports_unicode_filenames (*dans le module os.path*), 478
SupportsAbs (*classe dans typing*), 1709
SupportsBytes (*classe dans typing*), 1709
SupportsComplex (*classe dans typing*), 1709
SupportsFloat (*classe dans typing*), 1710
SupportsIndex (*classe dans typing*), 1710
SupportsInt (*classe dans typing*), 1710
SupportsRound (*classe dans typing*), 1710
suppress() (*dans le module contextlib*), 1993
SuppressCrashReport (*classe dans test.support*), 1855
surrogateescape
    error handler's name, 192
surrogatepass
    error handler's name, 192
SW_HIDE (*dans le module subprocess*), 1040
SWAP (*opcode*), 2166
swap_attr() (*dans le module test.support*), 1851
swap_item() (*dans le module test.support*), 1851
swapcase() (*méthode bytearray*), 77
swapcase() (*méthode bytes*), 77
swapcase() (*méthode str*), 62
Symbol (*classe dans symtable*), 2141
SymbolTable (*classe dans symtable*), 2140
SymbolTableType (*classe dans symtable*), 2139
symlink() (*dans le module os*), 703

symlink_to() (*méthode pathlib.Path*), 466
symmetric_difference() (*méthode frozenset*), 89
symmetric_difference_update() (*méthode frozenset*), 90
symtable
    module, 2139
symtable() (*dans le module symtable*), 2139
SYMTYPE (*dans le module tarfile*), 598
SYN (*dans le module curses.ascii*), 951
sync() (*dans le module os*), 703
sync() (*méthode dbm.dumb.dumbdbm*), 536
sync() (*méthode dbm.gnu.gdbm*), 534
sync() (*méthode shelve.Shelf*), 527
syncdown() (*méthode curses.window*), 936
synchronized() (*dans le module multiprocessing.sharedctypes*), 990
SyncManager (*classe dans multiprocessing.managers*), 993
syncok() (*méthode curses.window*), 936
syncup() (*méthode curses.window*), 936
SyntaxErr, 1383
SyntaxError, 112
SyntaxWarning, 116
sys
    module, 25, 1929
sys_version (*attribut http.server.BaseHTTPRequestHandler*), 1504
sysconf() (*dans le module os*), 728
sysconf_names (*dans le module os*), 728
sysconfig
    module, 1959
syslog
    module, 2210
syslog() (*dans le module syslog*), 2210
SysLogHandler (*classe dans logging.handlers*), 793
sys.monitoring
    module, 1955
system() (*dans le module os*), 721
system() (*dans le module platform*), 801
system_alias() (*dans le module platform*), 801
system_must_validate_cert() (*dans le module test.support*), 1852
SystemError, 113
SystemExit, 113
systemId (*attribut xml.dom.DocumentType*), 1378
SystemRandom (*classe dans random*), 388
SystemRandom (*classe dans secrets*), 659
SystemRoot, 1036

T

-T
    option de ligne de commande trace, 1899
-t
    option de ligne de commande calendar, 257
    option de ligne de commande tarfile, 609
    option de ligne de commande trace, 1899
    option de ligne de commande unittest-discover, 1755
    option de ligne de commande zipfile, 595
T_FMT (*dans le module locale*), 1563
T_FMT_AMPM (*dans le module locale*), 1563
--tab
    option de ligne de commande json, 1317
TAB (*dans le module curses.ascii*), 950
tab() (*méthode tkinter.ttk.Notebook*), 1647
TabError, 113
tableau de correspondances (*mapping en anglais*), **2238**
tabnanny
    module, 2151
tabs() (*méthode tkinter.ttk.Notebook*), 1647
tabsize (*attribut textwrap.TextWrapper*), 170
tabular
    data, 613
tag (*attribut xml.etree.ElementTree.Element*), 1366
tag_bind() (*méthode tkinter.ttk.Treeview*), 1654
tag_configure() (*méthode tkinter.ttk.Treeview*), 1654
tag_has() (*méthode tkinter.ttk.Treeview*), 1654
tagName (*attribut xml.dom.Element*), 1380
tail (*attribut xml.etree.ElementTree.Element*), 1366
take_snapshot() (*dans le module tracemalloc*), 1907
takewhile() (*dans le module itertools*), 418
tan() (*dans le module cmath*), 349
tan() (*dans le module math*), 345
tanh() (*dans le module cmath*), 349
tanh() (*dans le module math*), 346
tar_filter() (*dans le module tarfile*), 606
TarError, 597
tarfile
    module, 595
TarFile (*classe dans tarfile*), 599
target (*attribut xml.dom.ProcessingInstruction*), 1382
tarinfo (*attribut tarfile.FilterError*), 597
TarInfo (*classe dans tarfile*), 603
Task (*classe dans asyncio*), 1080
task_done() (*méthode asyncio.Queue*), 1103
task_done() (*méthode multiprocessing.JoinableQueue*), 982
task_done() (*méthode queue.Queue*), 1050
TaskGroup (*classe dans asyncio*), 1069
tau (*dans le module cmath*), 350
tau (*dans le module math*), 346
tb_locals (*attribut unittest.TestResult*), 1777
tbreak (*pdb command*), 1880
tcdrain() (*dans le module termios*), 2200
tcflow() (*dans le module termios*), 2200
tcflush() (*dans le module termios*), 2200
tcgetattr() (*dans le module termios*), 2200

tcgetpgrp() (*dans le module os*), 684
tcgetwinsize() (*dans le module termios*), 2200
Tcl() (*dans le module tkinter*), 1621
TCPServer (*classe dans socketserver*), 1493
TCSADRAIN (*dans le module termios*), 2200
TCSAFLUSH (*dans le module termios*), 2200
TCSANOW (*dans le module termios*), 2200
tcsendbreak() (*dans le module termios*), 2200
tcsetattr() (*dans le module termios*), 2200
tcsetpgrp() (*dans le module os*), 684
tcsetwinsize() (*dans le module termios*), 2200
tearDown() (*méthode unittest.TestCase*), 1761
tearDownClass() (*méthode unittest.TestCase*), 1762
tee() (*dans le module itertools*), 419
teleport() (*dans le module turtle*), 1580
tell() (*méthode io.IOBase*), 735
tell() (*méthode io.TextIOBase*), 740
tell() (*méthode io.TextIOWrapper*), 742
tell() (*méthode mmap.mmap*), 1247
tell() (*méthode sqlite3.Blob*), 555
tell() (*méthode wave.Wave_read*), 1550
tell() (*méthode wave.Wave_write*), 1551
telnetlib
    module, 2225
TEMP, 490
temp_cwd() (*dans le module test.support.os_helper*), 1860
temp_dir() (*dans le module test.support.os_helper*), 1861
temp_umask() (*dans le module test.support.os_helper*), 1861
tempdir (*dans le module tempfile*), 491
tempfile
    module, 487
template (*attribut string.Template*), 131
Template (*classe dans string*), 130
temporary
    file, 487
    file name, 487
temporary (*attribut bdb.Breakpoint*), 1870
TemporaryDirectory (*classe dans tempfile*), 489
TemporaryFile() (*dans le module tempfile*), 487
teredo (*attribut ipaddress.IPv6Address*), 1538
TERM, 929
termattrs() (*dans le module curses*), 929
terminal_size (*classe dans os*), 685
terminate() (*méthode asyncio.subprocess.Process*), 1100
terminate() (*méthode asyncio.SubprocessTransport*), 1138
terminate() (*méthode multiprocessing.pool.Pool*), 1000
terminate() (*méthode multiprocessing.Process*), 978
terminate() (*méthode subprocess.Popen*), 1038
terminator (*attribut logging.StreamHandler*), 787
termios
    module, 2200
termname() (*dans le module curses*), 929
test
    module, 1844
--test
    option de ligne de commande tarfile, 609
    option de ligne de commande zipfile, 595
test (*attribut doctest.DocTestFailure*), 1750
test (*attribut doctest.UnexpectedException*), 1750
TEST_DATA_DIR (*dans le module test.support*), 1848
TEST_HOME_DIR (*dans le module test.support*), 1848
TEST_HTTP_URL (*dans le module test.support*), 1849
TEST_SUPPORT_DIR (*dans le module test.support*), 1848
TestCase (*classe dans unittest*), 1761
TestFailed, 1847
testfile() (*dans le module doctest*), 1739
TESTFN (*dans le module test.support.os_helper*), 1859
TESTFN_NONASCII (*dans le module test.support.os_helper*), 1859
TESTFN_UNDECODABLE (*dans le module test.support.os_helper*), 1859
TESTFN_UNENCODABLE (*dans le module test.support.os_helper*), 1859
TESTFN_UNICODE (*dans le module test.support.os_helper*), 1859
TestLoader (*classe dans unittest*), 1774
testMethodPrefix (*attribut unittest.TestLoader*), 1776
testmod() (*dans le module doctest*), 1740
testNamePatterns (*attribut unittest.TestLoader*), 1776
test.regrtest
    module, 1846
TestResult (*classe dans unittest*), 1776
TestResults (*classe dans doctest*), 1745
testsource() (*dans le module doctest*), 1749
testsRun (*attribut unittest.TestResult*), 1777
TestSuite (*classe dans unittest*), 1773
test.support
    module, 1847
test.support.bytecode_helper
    module, 1858
test.support.import_helper
    module, 1861
test.support.os_helper
    module, 1859
test.support.script_helper
    module, 1857
test.support.socket_helper
    module, 1856
test.support.threading_helper
    module, 1858
test.support.warnings_helper
    module, 1862
testzip() (*méthode zipfile.ZipFile*), 589
text (*attribut ctypes.COMError*), 848
text (*attribut SyntaxError*), 113
text (*attribut traceback.TracebackException*), 2014

text (*attribut xml.etree.ElementTree.Element*), 1366
Text (*classe dans typing*), 1720
text mode, 25
text_encoding() (*dans le module io*), 732
text_factory (*attribut sqlite3.Connection*), 552
Textbox (*classe dans curses.textpad*), 948
TextCalendar (*classe dans calendar*), 251
textdomain() (*dans le module gettext*), 1554
textdomain() (*dans le module locale*), 1569
textinput() (*dans le module turtle*), 1599
TextIO (*classe dans typing*), 1710
TextIOBase (*classe dans io*), 740
TextIOWrapper (*classe dans io*), 741
TextTestResult (*classe dans unittest*), 1778
TextTestRunner (*classe dans unittest*), 1779
textwrap
    module, 168
TextWrapper (*classe dans textwrap*), 170
TFD_CLOEXEC (*dans le module os*), 710
TFD_NONBLOCK (*dans le module os*), 710
TFD_TIMER_ABSTIME (*dans le module os*), 710
TFD_TIMER_CANCEL_ON_SET (*dans le module os*), 710
theme_create() (*méthode tkinter.ttk.Style*), 1658
theme_names() (*méthode tkinter.ttk.Style*), 1658
theme_settings() (*méthode tkinter.ttk.Style*), 1658
theme_use() (*méthode tkinter.ttk.Style*), 1658
THOUSEP (*dans le module locale*), 1564
Thread (*classe dans threading*), 959
thread() (*méthode imaplib.IMAP4*), 1480
thread_info (*dans le module sys*), 1953
thread_time() (*dans le module time*), 752
thread_time_ns() (*dans le module time*), 752
thread_unsafe() (*dans le module test.support*), 1853
threading
    module, 955
threading_cleanup() (*dans le module test.support.threading_helper*), 1858
threading_setup() (*dans le module test.support.threading_helper*), 1858
ThreadingHTTPServer (*classe dans http.server*), 1503
ThreadingMixIn (*classe dans socketserver*), 1495
ThreadingMock (*classe dans unittest.mock*), 1799
ThreadingTCPServer (*classe dans socketserver*), 1495
ThreadingUDPServer (*classe dans socketserver*), 1495
ThreadingUnixDatagramServer (*classe dans socketserver*), 1495
ThreadingUnixStreamServer (*classe dans socketserver*), 1495
ThreadPool (*classe dans multiprocessing.pool*), 1004
ThreadPoolExecutor (*classe dans concurrent.futures*), 1022
threads
    POSIX, 1056
threadsafety (*dans le module sqlite3*), 541
THURSDAY (*dans le module calendar*), 255
ticket_lifetime_hint (*attribut ssl.SSLSession*), 1222
tigetflag() (*dans le module curses*), 929
tigetnum() (*dans le module curses*), 929
tigetstr() (*dans le module curses*), 929
TILDE (*dans le module token*), 2145
tilt() (*dans le module turtle*), 1591
tiltangle() (*dans le module turtle*), 1591
time
    module, 743
time (*attribut ssl.SSLSession*), 1222
time (*attribut uuid.UUID*), 1490
time (*classe dans datetime*), 229
time() (*dans le module time*), 751
time() (*méthode asyncio.loop*), 1111
time() (*méthode datetime.datetime*), 223
Time2Internaldate() (*dans le module imaplib*), 1475
time_hi_version (*attribut uuid.UUID*), 1490
time_low (*attribut uuid.UUID*), 1490
time_mid (*attribut uuid.UUID*), 1490
time_ns() (*dans le module time*), 752
timedelta (*classe dans datetime*), 209
TimedRotatingFileHandler (*classe dans logging.handlers*), 790
timegm() (*dans le module calendar*), 255
timeit
    module, 1893
timeit() (*dans le module timeit*), 1894
timeit() (*méthode timeit.Timer*), 1895
timeout, 1166
timeout (*attribut socketserver.BaseServer*), 1497
timeout (*attribut ssl.SSLSession*), 1222
timeout (*attribut subprocess.TimeoutExpired*), 1030
Timeout (*classe dans asyncio*), 1075
timeout() (*dans le module asyncio*), 1074
timeout() (*méthode curses.window*), 936
timeout_at() (*dans le module asyncio*), 1075
TIMEOUT_MAX (*dans le module _thread*), 1058
TIMEOUT_MAX (*dans le module threading*), 958
TimeoutError, 116, 979, 1027, 1105
TimeoutExpired, 1030
Timer (*classe dans threading*), 967
Timer (*classe dans timeit*), 1894
timerfd_create() (*dans le module os*), 708
timerfd_gettime() (*dans le module os*), 710
timerfd_gettime_ns() (*dans le module os*), 710
timerfd_settime() (*dans le module os*), 709
timerfd_settime_ns() (*dans le module os*), 710
TimerHandle (*classe dans asyncio*), 1126
times() (*dans le module os*), 721
TIMESTAMP (*attribut py_compile.PycInvalidationMode*), 2155
timestamp() (*méthode datetime.datetime*), 225
timetuple() (*méthode datetime.date*), 215
timetuple() (*méthode datetime.datetime*), 224
timetz() (*méthode datetime.datetime*), 223
timezone (*classe dans datetime*), 239
timezone (*dans le module time*), 755
--timing

option de ligne de commande trace, 1900
title() (*dans le module turtle*), 1602
title() (*méthode bytearray*), 77
title() (*méthode bytes*), 77
title() (*méthode str*), 62
Tk, 1619
tk (*attribut tkinter.Tk*), 1621
Tk (*classe dans tkinter*), 1621
Tkinter, 1619
tkinter
module, 1619
tkinter.colorchooser
module, 1632
tkinter.commondialog
module, 1637
tkinter.dnd
module, 1639
tkinter.filedialog
module, 1634
tkinter.font
module, 1633
tkinter.messagebox
module, 1637
tkinter.scrolledtext
module, 1639
tkinter.simpledialog
module, 1634
tkinter.ttk
module, 1640
TLS, 1189
TLSv1 (*attribut ssl.TLSVersion*), 1201
TLSv1_1 (*attribut ssl.TLSVersion*), 1201
TLSv1_2 (*attribut ssl.TLSVersion*), 1201
TLSv1_3 (*attribut ssl.TLSVersion*), 1202
TLSVersion (*classe dans ssl*), 1201
tm_gmtoff (*attribut time.struct_time*), 751
tm_hour (*attribut time.struct_time*), 751
tm_isdst (*attribut time.struct_time*), 751
tm_mday (*attribut time.struct_time*), 751
tm_min (*attribut time.struct_time*), 751
tm_mon (*attribut time.struct_time*), 751
tm_sec (*attribut time.struct_time*), 751
tm_wday (*attribut time.struct_time*), 751
tm_yday (*attribut time.struct_time*), 751
tm_year (*attribut time.struct_time*), 751
tm_zone (*attribut time.struct_time*), 751
TMP, 490
TMPDIR, 490
TO_BOOL (*opcode*), 2167
to_bytes() (*méthode int*), 41
to_eng_string() (*méthode decimal.Context*), 369
to_eng_string() (*méthode decimal.Decimal*), 362
to_integral() (*méthode decimal.Decimal*), 362
to_integral_exact() (*méthode decimal.Context*), 369
to_integral_exact() (*méthode decimal.Decimal*), 362
to_integral_value() (*méthode decimal.Decimal*), 362
to_sci_string() (*méthode decimal.Context*), 369
to_thread() (*dans le module asyncio*), 1078
ToASCII() (*dans le module encodings.idna*), 205
tobuf() (*méthode tarfile.TarInfo*), 603
tobytes() (*méthode array.array*), 291
tobytes() (*méthode memoryview*), 83
today() (*méthode de la classe datetime.date*), 213
today() (*méthode de la classe datetime.datetime*), 218
tofile() (*méthode array.array*), 291
tok_name (*dans le module token*), 2143
token
module, 2143
token (*attribut shlex.shlex*), 1616
Token (*classe dans contextvars*), 1053
token_bytes() (*dans le module secrets*), 659
token_hex() (*dans le module secrets*), 659
token_urlsafe() (*dans le module secrets*), 659
TokenError, 2149
tokenize
module, 2147
tokenize() (*dans le module tokenize*), 2148
tolist() (*méthode array.array*), 291
tolist() (*méthode memoryview*), 83
TOMLDecodeError, 640
tomllib
module, 639
toordinal() (*méthode datetime.date*), 216
toordinal() (*méthode datetime.datetime*), 225
top() (*méthode curses.panel.Panel*), 954
top() (*méthode poplib.POP3*), 1473
top_panel() (*dans le module curses.panel*), 953
--top-level-directory
option de ligne de commande unittest-discover, 1755
TopologicalSorter (*classe dans graphlib*), 331
toprettyxml() (*méthode xml.dom.minidom.Node*), 1386
toreadonly() (*méthode memoryview*), 84
tostring() (*dans le module xml.etree.ElementTree*), 1364
tostringlist() (*dans le module xml.etree.ElementTree*), 1364
total() (*méthode collections.Counter*), 262
total_changes (*attribut sqlite3.Connection*), 552
total_nframe (*attribut tracemalloc.Traceback*), 1910
total_ordering() (*dans le module functools*), 429
total_seconds() (*méthode datetime.timedelta*), 212
touch() (*méthode pathlib.Path*), 466
touchline() (*méthode curses.window*), 937
touchwin() (*méthode curses.window*), 937
ToUnicode() (*dans le module encodings.idna*), 205
tounicode() (*méthode array.array*), 291
towards() (*dans le module turtle*), 1583
toxml() (*méthode xml.dom.minidom.Node*), 1386
tparm() (*dans le module curses*), 929
trace

module, 1898
--trace
option de ligne de commande trace, 1899
Trace (*classe dans trace*), 1900
Trace (*classe dans tracemalloc*), 1910
trace function, 957, 1941, 1949
trace() (*dans le module inspect*), 2041
trace_dispatch() (*méthode bdb.Bdb*), 1870
traceback
module, 2010
object, 1934, 2010
traceback (*attribut tracemalloc.Statistic*), 1909
traceback (*attribut tracemalloc.StatisticDiff*), 1910
traceback (*attribut tracemalloc.Trace*), 1910
Traceback (*classe dans inspect*), 2039
Traceback (*classe dans tracemalloc*), 1910
traceback_limit (*attribut tracemalloc.Snapshot*), 1909
traceback_limit (*attribut wsgiref.handlers.BaseHandler*), 1423
TracebackException (*classe dans traceback*), 2013
tracebacklimit (*dans le module sys*), 1953
TracebackType (*classe dans types*), 303
tracemalloc
module, 1901
tracer() (*dans le module turtle*), 1597
traces (*attribut tracemalloc.Snapshot*), 1909
--trackcalls
option de ligne de commande trace, 1899
tranche, **2243**
transfercmd() (*méthode ftplib.FTP*), 1468
transient_internet() (*dans le module test.support.socket_helper*), 1856
translate() (*dans le module fnmatch*), 496
translate() (*dans le module glob*), 494
translate() (*méthode bytearray*), 71
translate() (*méthode bytes*), 71
translate() (*méthode str*), 62
translation() (*dans le module gettext*), 1555
transport (*attribut asyncio.StreamWriter*), 1088
Transport (*classe dans asyncio*), 1135
Transport Layer Security, 1189
Traversable (*classe dans importlib.abc*), 2073
Traversable (*classe dans importlib.resources.abc*), 2089
TraversableResources (*classe dans importlib.abc*), 2074
TraversableResources (*classe dans importlib.resources.abc*), 2089
TreeBuilder (*classe dans xml.etree.ElementTree*), 1370
Treeview (*classe dans tkinter.ttk*), 1651
triangular() (*dans le module random*), 387
tries (*attribut doctest.DocTestRunner*), 1747
True, 37, 46
true, 37
True (*variable de base*), 35
truediv() (*dans le module operator*), 439
trunc() (*dans le module math*), 341
trunc() (*in module math*), 40
truncate() (*dans le module os*), 703
truncate() (*méthode io.IOBase*), 735
truth
value, 37
truth() (*dans le module operator*), 438
try
statement, 107
Try (*classe dans ast*), 2121
TryStar (*classe dans ast*), 2121
ttk, 1640
tty
I/O control, 2200
module, 2201
ttyname() (*dans le module os*), 684
TUESDAY (*dans le module calendar*), 255
tuple
object, 48, 50
Tuple (*classe dans ast*), 2108
tuple (*classe de base*), 50
Tuple (*dans le module typing*), 1719
turtle
module, 1571
Turtle (*classe dans turtle*), 1602
turtledemo
module, 1606
turtles() (*dans le module turtle*), 1601
TurtleScreen (*classe dans turtle*), 1602
turtlesize() (*dans le module turtle*), 1590
typage canard (*duck-typing*), **2233**
type, **2244**
Boolean, 9
built-in function, 103
object, 32
operations on dictionary, 90
operations on list, 49
union, 100
--type
option de ligne de commande calendar, 257
type (*attribut optparse.Option*), 907
type (*attribut socket.socket*), 1185
type (*attribut tarfile.TarInfo*), 603
type (*attribut urllib.request.Request*), 1431
Type (*classe dans typing*), 1719
type (*classe de base*), 32
type générique, **2235**
TYPE_ALIAS (*attribut symtable.SymbolTableType*), 2140
type_check_only() (*dans le module typing*), 1715
TYPE_CHECKER (*attribut optparse.Option*), 917
TYPE_CHECKING (*dans le module typing*), 1718
type_comment (*attribut ast.arg*), 2131
type_comment (*attribut ast.Assign*), 2115
type_comment (*attribut ast.For*), 2119
type_comment (*attribut ast.FunctionDef*), 2130

type_comment (*attribut ast.With*), 2122
TYPE_COMMENT (*dans le module token*), 2146
TYPE_IGNORE (*dans le module token*), 2146
TYPE_PARAMETERS (*attribut symtable.SymbolTableType*), 2140
TYPE_VARIABLE (*attribut symtable.SymbolTableType*), 2140
typeahead() (*dans le module curses*), 930
TypeAlias (*classe dans ast*), 2117
TypeAlias (*dans le module typing*), 1686
TypeAliasType (*classe dans typing*), 1701
typecode (*attribut array.array*), 290
typecodes (*dans le module array*), 289
TYPED_ACTIONS (*attribut optparse.Option*), 919
typed_subpart_iterator() (*dans le module email.iterators*), 1307
TypedDict (*classe dans typing*), 1706
TypeError, 113
TypeGuard (*dans le module typing*), 1693
TypeIs (*dans le module typing*), 1692
types
    built-in, 37
    immutable sequence, 48
    module, 103, 299
    mutable sequence, 49
    operations on integer, 40
    operations on mapping, 90
    operations on numeric, 39
    operations on sequence, 47, 49
TYPES (*attribut optparse.Option*), 917
Types de données des options *Tk*, 1629
types_map (*attribut mimetypes.MimeTypes*), 1338
types_map (*dans le module mimetypes*), 1337
types_map_inv (*attribut mimetypes.MimeTypes*), 1338
TypeVar (*classe dans ast*), 2129
TypeVar (*classe dans typing*), 1695
TypeVarTuple (*classe dans ast*), 2129
TypeVarTuple (*classe dans typing*), 1697
typing
    module, 1671
TZ, 752, 753
tzinfo (*attribut datetime.datetime*), 222
tzinfo (*attribut datetime.time*), 230
tzinfo (*classe dans datetime*), 233
tzname (*dans le module time*), 755
tzname() (*méthode datetime.datetime*), 224
tzname() (*méthode datetime.time*), 232
tzname() (*méthode datetime.timezone*), 240
tzname() (*méthode datetime.tzinfo*), 234
TZPATH (*dans le module zoneinfo*), 249
tzset() (*dans le module time*), 752

U

-u
    option de ligne de commande timeit, 1896
    option de ligne de commande uuid, 1492
U (*dans le module re*), 141
UAdd (*classe dans ast*), 2110
ucd_3_2_0 (*dans le module unicodedata*), 173
udata (*attribut select.kevent*), 1231
UDPServer (*classe dans socketserver*), 1494
UF_APPEND (*dans le module stat*), 483
UF_COMPRESSED (*dans le module stat*), 483
UF_DATAVAULT (*dans le module stat*), 483
UF_HIDDEN (*dans le module stat*), 483
UF_IMMUTABLE (*dans le module stat*), 483
UF_NODUMP (*dans le module stat*), 483
UF_NOUNLINK (*dans le module stat*), 483
UF_OPAQUE (*dans le module stat*), 483
UF_SETTABLE (*dans le module stat*), 482
UF_TRACKED (*dans le module stat*), 483
uid (*attribut tarfile.TarInfo*), 604
UID (*classe dans plistlib*), 643
uid() (*méthode imaplib.IMAP4*), 1481
uidl() (*méthode poplib.POP3*), 1473
ulp() (*dans le module math*), 343
umask() (*dans le module os*), 671
unalias (*pdb command*), 1884
uname (*attribut tarfile.TarInfo*), 604
uname() (*dans le module os*), 672
uname() (*dans le module platform*), 802
UNARY_INVERT (*opcode*), 2167
UNARY_NEGATIVE (*opcode*), 2166
UNARY_NOT (*opcode*), 2166
UnaryOp (*classe dans ast*), 2110
UnboundLocalError, 114
unbuffered I/O, 25
UNC paths
    and os.makedirs(), 692
uncancel() (*méthode asyncio.Task*), 1083
UNCHECKED_HASH (*attribut py_compile.PycInvalidationMode*), 2155
unconsumed_tail (*attribut zlib.Decompress*), 570
unctrl() (*dans le module curses*), 930
unctrl() (*dans le module curses.ascii*), 953
Underflow (*classe dans decimal*), 371
undisplay (*pdb command*), 1883
undo() (*dans le module turtle*), 1582
undobufferentries() (*dans le module turtle*), 1594
undoc_header (*attribut cmd.Cmd*), 1610
unescape() (*dans le module html*), 1347
unescape() (*dans le module xml.sax.saxutils*), 1398
UnexpectedException, 1750
unexpectedSuccesses (*attribut unittest.TestResult*), 1777
unfreeze() (*dans le module gc*), 2024
unget_wch() (*dans le module curses*), 930
ungetch() (*dans le module curses*), 930
ungetch() (*dans le module msvcrt*), 2184
ungetmouse() (*dans le module curses*), 930
ungetwch() (*dans le module msvcrt*), 2184
unhexlify() (*dans le module binascii*), 1344
Unicode, 172, 189
    database, 172
UNICODE (*dans le module re*), 141

unicodedata
    module, 172
UnicodeDecodeError, 114
UnicodeEncodeError, 114
UnicodeError, 114
UnicodeTranslateError, 114
UnicodeWarning, 117
unidata_version (*dans le module unicodedata*), 173
unified_diff() (*dans le module difflib*), 159
uniform() (*dans le module random*), 387
UnimplementedFileMode, 1459
Union
    object, 100
union
    type, 100
Union (*classe dans ctypes*), 846
Union (*dans le module typing*), 1686
union() (*méthode frozenset*), 89
UnionType (*classe dans types*), 303
UNIQUE (*attribut enum.EnumCheck*), 326
unique() (*dans le module enum*), 329
--unit
    option de ligne de commande timeit, 1896
unittest
    module, 1751
unittest.mock
    module, 1783
universal newlines
    bytearray.splitlines method, 76
    bytes.splitlines method, 76
    csv.reader function, 613
    importlib.abc.InspectLoader.get_source method, 2070
    io.IncrementalNewlineDecoder class, 743
    io.TextIOWrapper class, 741
    open() built-in function, 25
    str.splitlines method, 60
    subprocess module, 1031
UNIX
    file control, 2204
    I/O control, 2204
unix_dialect (*classe dans csv*), 616
unix_shell (*dans le module test.support*), 1847
UnixDatagramServer (*classe dans socketserver*), 1494
UnixStreamServer (*classe dans socketserver*), 1494
unknown (*attribut uuid.SafeUUID*), 1489
unknown_decl() (*méthode html.parser.HTMLParser*), 1350
unknown_open() (*méthode urllib.request.BaseHandler*), 1434
unknown_open() (*méthode urllib.request.UnknownHandler*), 1438
UnknownHandler (*classe dans urllib.request*), 1431
UnknownProtocol, 1459
UnknownTransferEncoding, 1459
unlink() (*dans le module os*), 703
unlink() (*dans le module test.support.os_helper*), 1861
unlink() (*méthode multiprocessing.shared_memory.SharedMemory*), 1015
unlink() (*méthode pathlib.Path*), 468
unlink() (*méthode xml.dom.minidom.Node*), 1386
unload() (*dans le module test.support.import_helper*), 1862
unlock() (*méthode mailbox.Babyl*), 1326
unlock() (*méthode mailbox.Mailbox*), 1320
unlock() (*méthode mailbox.Maildir*), 1322
unlock() (*méthode mailbox.mbox*), 1323
unlock() (*méthode mailbox.MH*), 1324
unlock() (*méthode mailbox.MMDF*), 1326
unlockpt() (*dans le module os*), 684
UNNAMED_SECTION (*dans le module configparser*), 637
UnnamedSectionDisabledError, 639
Unpack (*dans le module typing*), 1694
unpack() (*dans le module struct*), 182
unpack() (*méthode struct.Struct*), 188
unpack_archive() (*dans le module shutil*), 504
UNPACK_EX (*opcode*), 2171
unpack_from() (*dans le module struct*), 182
unpack_from() (*méthode struct.Struct*), 188
UNPACK_SEQUENCE (*opcode*), 2171
unparse() (*dans le module ast*), 2134
unparsedEntityDecl() (*méthode xml.sax.handler.DTDHandler*), 1397
UnparsedEntityDeclHandler() (*méthode xml.parsers.expat.xmlparser*), 1407
Unpickler (*classe dans pickle*), 513
UnpicklingError, 512
unquote() (*dans le module email.utils*), 1304
unquote() (*dans le module urllib.parse*), 1450
unquote_plus() (*dans le module urllib.parse*), 1450
unquote_to_bytes() (*dans le module urllib.parse*), 1450
unraisablehook() (*dans le module sys*), 1953
unregister() (*dans le module atexit*), 2009
unregister() (*dans le module codecs*), 190
unregister() (*dans le module faulthandler*), 1875
unregister() (*méthode select.devpoll*), 1226
unregister() (*méthode select.epoll*), 1227
unregister() (*méthode selectors.BaseSelector*), 1232
unregister() (*méthode select.poll*), 1228
unregister_archive_format() (*dans le module shutil*), 504
unregister_dialect() (*dans le module csv*), 614
unregister_unpack_format() (*dans le module shutil*), 505
unsafe (*attribut uuid.SafeUUID*), 1488
unselect() (*méthode imaplib.IMAP4*), 1481
unset() (*méthode test.support.os_helper.EnvironmentVarGuard*), 1860
unsetenv() (*dans le module os*), 672
unshare() (*dans le module os*), 672
UnstructuredHeader (*classe dans email.headerregistry*), 1274
unsubscribe() (*méthode imaplib.IMAP4*), 1481

UnsupportedOperation, 447, 732
until (*pdb command*), 1881
untokenize() (*dans le module tokenize*), 2148
untouchwin() (*méthode curses.window*), 937
unused_data (*attribut bz2.BZ2Decompressor*), 577
unused_data (*attribut lzma.LZMADecompressor*), 582
unused_data (*attribut zlib.Decompress*), 569
unverifiable (*attribut urllib.request.Request*), 1432
unwrap() (*dans le module inspect*), 2038
unwrap() (*dans le module urllib.parse*), 1448
unwrap() (*méthode ssl.SSLSocket*), 1205
up (*pdb command*), 1880
up() (*dans le module turtle*), 1585
update() (*dans le module turtle*), 1597
update() (*méthode collections.Counter*), 263
update() (*méthode dict*), 93
update() (*méthode frozenset*), 90
update() (*méthode hashlib.hash*), 647
update() (*méthode hmac.HMAC*), 657
update() (*méthode http.cookies.Morsel*), 1511
update() (*méthode mailbox.Mailbox*), 1320
update() (*méthode mailbox.Maildir*), 1322
update() (*méthode trace.CoverageResults*), 1900
update_abstractmethods() (*dans le module abc*), 2008
update_authenticated() (*méthode urllib.request.HTTPPasswordMgrWithPriorAuth*), 1436
update_lines_cols() (*dans le module curses*), 930
update_panels() (*dans le module curses.panel*), 953
update_visible() (*méthode mailbox.BabylMessage*), 1332
update_wrapper() (*dans le module functools*), 436
upgrade_dependencies() (*méthode venv.EnvBuilder*), 1919
upper() (*méthode bytearray*), 78
upper() (*méthode bytes*), 78
upper() (*méthode str*), 62
urandom() (*dans le module os*), 729
URL, 1443, 1452, 1502
    parsing, 1443
    relative, 1443
url (*attribut http.client.HTTPResponse*), 1463
url (*attribut urllib.error.HTTPError*), 1452
url (*attribut urllib.response.addinfourl*), 1442
url (*attribut xmlrpc.client.ProtocolError*), 1526
url2pathname() (*dans le module urllib.request*), 1428
urlcleanup() (*dans le module urllib.request*), 1442
urldefrag() (*dans le module urllib.parse*), 1447
urlencode() (*dans le module urllib.parse*), 1450
URLError, 1452
urljoin() (*dans le module urllib.parse*), 1447
urllib
    module, 1426
urllib.error
    module, 1452
urllib.parse
    module, 1443
urllib.request
    module, 1426, 1457
urllib.response
    module, 1442
urllib.robotparser
    module, 1452
urlopen() (*dans le module urllib.request*), 1426
urlparse() (*dans le module urllib.parse*), 1443
urlretrieve() (*dans le module urllib.request*), 1441
urlsafe_b64decode() (*dans le module base64*), 1340
urlsafe_b64encode() (*dans le module base64*), 1340
urlsplit() (*dans le module urllib.parse*), 1446
urlunparse() (*dans le module urllib.parse*), 1446
urlunsplit() (*dans le module urllib.parse*), 1447
urn (*attribut uuid.UUID*), 1490
US (*dans le module curses.ascii*), 951
use_default_colors() (*dans le module curses*), 930
use_env() (*dans le module curses*), 930
use_rawinput (*attribut cmd.Cmd*), 1610
use_tool_id() (*dans le module sys.monitoring*), 1955
UseForeignDTD() (*méthode xml.parsers.expat.xmlparser*), 1404
USER, 921
user
    effective id, 667
    id, 669
    id, setting, 671
user() (*méthode poplib.POP3*), 1472
USER_BASE (*dans le module site*), 2051
user_call() (*méthode bdb.Bdb*), 1871
user_exception() (*méthode bdb.Bdb*), 1872
user_line() (*méthode bdb.Bdb*), 1872
user_return() (*méthode bdb.Bdb*), 1872
USER_SITE (*dans le module site*), 2051
--user-base
    option de ligne de commande site, 2052
usercustomize
    module, 2050
UserDict (*classe dans collections*), 275
UserList (*classe dans collections*), 276
USERNAME, 474, 668, 921
username (*attribut email.headerregistry.Address*), 1278
USERPROFILE, 474
userptr() (*méthode curses.panel.Panel*), 954
--user-site
    option de ligne de commande site, 2052
UserString (*classe dans collections*), 276
UserWarning, 116
USTAR_FORMAT (*dans le module tarfile*), 598
USub (*classe dans ast*), 2110
UTC, 744
utc (*attribut datetime.timezone*), 240
UTC (*dans le module datetime*), 208
utcfromtimestamp() (*méthode de la classe datetime.datetime*), 219
utcnow() (*méthode de la classe datetime.datetime*), 219
utcoffset() (*méthode datetime.datetime*), 224
utcoffset() (*méthode datetime.time*), 232

utcoffset() (*méthode datetime.timezone*), 240
utcoffset() (*méthode datetime.tzinfo*), 233
utctimetuple() (*méthode datetime.datetime*), 224
utf8 (*attribut email.policy.EmailPolicy*), 1269
utf8() (*méthode poplib.POP3*), 1473
utf8_enabled (*attribut imaplib.IMAP4*), 1481
utf8_mode (*attribut sys.flags*), 1936
utime() (*dans le module os*), 704
uu
    module, 2225
uuid
    module, 1488
--uuid
    option de ligne de commande uuid, 1492
UUID (*classe dans uuid*), 1489
uuid1, 1491
uuid1() (*dans le module uuid*), 1491
uuid3, 1491
uuid3() (*dans le module uuid*), 1491
uuid4, 1491
uuid4() (*dans le module uuid*), 1491
uuid5, 1491
uuid5() (*dans le module uuid*), 1491
uuid8, 1491
uuid8() (*dans le module uuid*), 1491

V

-v
    option de ligne de commande python--m-sqlite3-[-h]-[-v]-[filename]-[sql], 558
    option de ligne de commande tarfile, 609
    option de ligne de commande timeit, 1896
    option de ligne de commande unittest-discover, 1755
v4_int_to_packed() (*dans le module ipaddress*), 1546
v6_int_to_packed() (*dans le module ipaddress*), 1547
valid_signals() (*dans le module signal*), 1238
validator() (*dans le module wsgiref.validate*), 1420
value
    truth, 37
VALUE (*attribut annotationlib.Format*), 2046
value (*attribut ctypes._SimpleCData*), 843
value (*attribut enum.Enum*), 319
value (*attribut http.cookiejar.Cookie*), 1519
value (*attribut http.cookies.Morsel*), 1511
value (*attribut StopIteration*), 112
value (*attribut xml.dom.Attr*), 1381
Value() (*dans le module multiprocessing*), 988
Value() (*dans le module multiprocessing.sharedctypes*), 990
Value() (*méthode multiprocessing.managers.SyncManager*), 993
value_decode() (*méthode http.cookies.BaseCookie*), 1509
value_encode() (*méthode http.cookies.BaseCookie*), 1509
value_to_string() (*dans le module annotationlib*), 2049
VALUE_WITH_FAKE_GLOBALS (*attribut annotationlib.Format*), 2046
ValueError, 114
valuerefs() (*méthode weakref.WeakValueDictionary*), 294
values
    Boolean, 46
Values (*classe dans optparse*), 907
values() (*méthode contextvars.Context*), 1055
values() (*méthode dict*), 93
values() (*méthode email.message.EmailMessage*), 1253
values() (*méthode email.message.Message*), 1291
values() (*méthode mailbox.Mailbox*), 1319
values() (*méthode types.MappingProxyType*), 304
ValuesView (*classe dans collections.abc*), 280
ValuesView (*classe dans typing*), 1722
var (*attribut contextvars.Token*), 1054
variable de classe, **2231**
variable de contexte, **2232**
variable d'environnement
    BROWSER, 1413, 1414
    COLUMNS, 930
    COMSPEC, 721, 1033
    DISPLAY, 1621
    HOME, 474, 1621
    HOMEDRIVE, 474
    HOMEPATH, 474
    IDLE*STARTUP, 1666
    LANG, 1553, 1555, 1561, 1565
    LANGUAGE, 1553, 1554
    LC_ALL, 1553, 1554
    LC_MESSAGES, 1553, 1555
    LINES, 925, 930
    LOGNAME, 668, 921
    MANPAGER, 1725
    no_proxy, 1430
    PAGER, 1725
    PATH, 471, 502, 712, 713, 718, 719, 728, 1032, 1413, 1917, 2050
    PATHEXT, 502
    PYTHON_CPU_COUNT, 727, 983
    PYTHON_DOM, 1375
    PYTHON_GIL, 2236
    PYTHONASYNCIODEBUG, 1124, 1159, 1726
    PYTHONBREAKPOINT, 10, 1932
    PYTHONCASEOK, 34
    PYTHONCOERCECLOCALE, 665
    PYTHONDEVMODE, 1725
    PYTHONDONDONTWRITEBYTECODE, 1933
    PYTHONFAULTHANDLER, 1726, 1874
    PYTHONHOME, 1857, 2097, 2098

PYTHONINTMAXSTRDIGITS, 105, 1944
PYTHONIOENCODING, 664, 1952
PYTHONLEGACYWINDOWSFSENCODING, 1951
PYTHONLEGACYWINDOWSSTDIO, 1952
PYTHONMALLOC, 1726
PYTHONNOUSERSITE, 2051
PYTHONPATH, 1857, 1946, 2097
PYTHONPLATLIBDIR, 2098
PYTHONPYCACHEPREFIX, 1933
PYTHONSAFEPATH, 1946, 2227
PYTHONSTARTUP, 178, 1062, 1666, 1944, 2051
PYTHONTRACEMALLOC, 1901, 1906
PYTHONTZPATH, 246, 250
PYTHONUNBUFFERED, 1952
PYTHONUSERBASE, 2051
PYTHONUSERSITE, 1857
PYTHONUTF8, 664, 665, 1952
PYTHONWARNDEFAULTENCODING, 732
PYTHONWARNINGS, 1726, 1972, 1973
SOURCE_DATE_EPOCH, 2154, 2155, 2157
SSLKEYLOGFILE, 1191, 1192
SystemRoot, 1036
TEMP, 490
TERM, 929
TMP, 490
TMPDIR, 490
TZ, 752, 753
USER, 921
USERNAME, 474, 668, 921
USERPROFILE, 474
variance (*attribut statistics.NormalDist*), 404
variance() (*dans le module statistics*), 401
variant (*attribut uuid.UUID*), 1490
vars()
built-in function, 32
vbar (*attribut tkinter.scrolledtext.ScrolledText*), 1639
VBAR (*dans le module token*), 2144
VBAREQUAL (*dans le module token*), 2145
VC_ASSEMBLY_PUBLICKEYTOKEN (*dans le module msvcrt*), 2185
Vec2D (*classe dans turtle*), 1603
venv
module, 1915
--verbose
option de ligne de commande tarfile, 609
option de ligne de commande timeit, 1896
option de ligne de commande unittest-discover, 1755
verbose (*attribut sys.flags*), 1936
VERBOSE (*dans le module re*), 141
verbose (*dans le module tabnanny*), 2152
verbose (*dans le module test.support*), 1847
verify() (*dans le module enum*), 330
verify() (*méthode smtplib.SMTP*), 1485
VERIFY_ALLOW_PROXY_CERTS (*dans le module ssl*), 1196
verify_client_post_handshake() (*méthode ssl.SSLSocket*), 1205
verify_code (*attribut ssl.SSLCertVerificationError*), 1193
VERIFY_CRL_CHECK_CHAIN (*dans le module ssl*), 1196
VERIFY_CRL_CHECK_LEAF (*dans le module ssl*), 1196
VERIFY_DEFAULT (*dans le module ssl*), 1196
verify_flags (*attribut ssl.SSLContext*), 1213
verify_generated_headers (*attribut email.policy.Policy*), 1267
verify_message (*attribut ssl.SSLCertVerificationError*), 1193
verify_mode (*attribut ssl.SSLContext*), 1213
verify_request() (*méthode socketserver.BaseServer*), 1498
VERIFY_X509_PARTIAL_CHAIN (*dans le module ssl*), 1196
VERIFY_X509_STRICT (*dans le module ssl*), 1196
VERIFY_X509_TRUSTED_FIRST (*dans le module ssl*), 1196
VerifyFlags (*classe dans ssl*), 1196
VerifyMode (*classe dans ssl*), 1196
verrou global de l'interpréteur, **2236**
--version
option de ligne de commande python--m-sqlite3-[-h]-[-v]-[filename]-[sql], 558
option de ligne de commande trace, 1899
version (*attribut email.headerregistry.MIMEVersionHeader*), 1276
version (*attribut http.client.HTTPResponse*), 1463
version (*attribut http.cookiejar.Cookie*), 1519
version (*attribut http.cookies.Morsel*), 1510
version (*attribut ipaddress.IPv4Address*), 1535
version (*attribut ipaddress.IPv4Network*), 1540
version (*attribut ipaddress.IPv6Address*), 1538
version (*attribut ipaddress.IPv6Network*), 1543
version (*attribut sys.thread_info*), 1953
version (*attribut uuid.UUID*), 1490
version (*dans le module curses*), 937
version (*dans le module marshal*), 530
version (*dans le module sys*), 1954
version() (*dans le module ensurepip*), 1914
version() (*dans le module importlib.metadata*), 2093
version() (*dans le module platform*), 802
version() (*méthode ssl.SSLSocket*), 1205
version_info (*dans le module sys*), 1954
version_string() (*méthode http.server.BaseHTTPRequestHandler*), 1506
vformat() (*méthode string.Formatter*), 122
virtual
Environments, 1915
visit() (*méthode ast.NodeVisitor*), 2136
visit_Constant() (*méthode ast.NodeVisitor*), 2136
vline() (*méthode curses.window*), 937
voidcmd() (*méthode ftplib.FTP*), 1467

volume (*attribut zipfile.ZipInfo*), 594
vonmisesvariate() (*dans le module random*), 388
VT (*dans le module curses.ascii*), 950
vue de dictionnaire, **2233**

W

-w
    option de ligne de commande calendar, 258
W_OK (*dans le module os*), 687
wait() (*dans le module asyncio*), 1076
wait() (*dans le module concurrent.futures*), 1027
wait() (*dans le module multiprocessing.connection*), 1002
wait() (*dans le module os*), 721
wait() (*méthode asyncio.Barrier*), 1097
wait() (*méthode asyncio.Condition*), 1095
wait() (*méthode asyncio.Event*), 1093
wait() (*méthode asyncio.subprocess.Process*), 1100
wait() (*méthode multiprocessing.pool.AsyncResult*), 1000
wait() (*méthode subprocess.Popen*), 1037
wait() (*méthode threading.Barrier*), 968
wait() (*méthode threading.Condition*), 964
wait() (*méthode threading.Event*), 967
wait3() (*dans le module os*), 722
wait4() (*dans le module os*), 722
wait_closed() (*méthode asyncio.Server*), 1127
wait_closed() (*méthode asyncio.StreamWriter*), 1089
wait_for() (*dans le module asyncio*), 1076
wait_for() (*méthode asyncio.Condition*), 1095
wait_for() (*méthode threading.Condition*), 964
wait_process() (*dans le module test.support*), 1852
wait_threads_exit() (*dans le module test.support.threading_helper*), 1858
wait_until_any_call_with() (*méthode unittest.mock.ThreadingMock*), 1800
wait_until_called() (*méthode unittest.mock.ThreadingMock*), 1799
waitid() (*dans le module os*), 721
waitpid() (*dans le module os*), 722
waitstatus_to_exitcode() (*dans le module os*), 724
walk() (*dans le module ast*), 2136
walk() (*dans le module os*), 704
walk() (*méthode email.message.EmailMessage*), 1255
walk() (*méthode email.message.Message*), 1295
walk() (*méthode pathlib.Path*), 465
walk_packages() (*dans le module pkgutil*), 2061
walk_stack() (*dans le module traceback*), 2013
walk_tb() (*dans le module traceback*), 2013
want (*attribut doctest.Example*), 1744
warn() (*dans le module warnings*), 1976
warn_default_encoding (*attribut sys.flags*), 1936
warn_explicit() (*dans le module warnings*), 1976
Warning, 116, 556
WARNING (*dans le module logging*), 762
WARNING (*dans le module tkinter.messagebox*), 1639
warning() (*dans le module logging*), 770
warning() (*méthode logging.Logger*), 760
warning() (*méthode xml.sax.handler.ErrorHandler*), 1397
warnings, 1971
    module, 1971
WarningsRecorder (*classe dans test.support.warnings_helper*), 1863
warnoptions (*dans le module sys*), 1954
wasSuccessful() (*méthode unittest.TestResult*), 1777
WatchedFileHandler (*classe dans logging.handlers*), 788
wave
    module, 1549
Wave_read (*classe dans wave*), 1550
Wave_write (*classe dans wave*), 1551
WCONTINUED (*dans le module os*), 723
WCOREDUMP() (*dans le module os*), 724
WeakKeyDictionary (*classe dans weakref*), 293
WeakMethod (*classe dans weakref*), 294
weakref
    module, 292
WeakSet (*classe dans weakref*), 294
WeakValueDictionary (*classe dans weakref*), 294
webbrowser
    module, 1413
WEDNESDAY (*dans le module calendar*), 255
weekday (*attribut calendar.IllegalWeekdayError*), 256
weekday() (*dans le module calendar*), 254
weekday() (*méthode datetime.date*), 216
weekday() (*méthode datetime.datetime*), 225
weekheader() (*dans le module calendar*), 254
weibullvariate() (*dans le module random*), 388
WEXITED (*dans le module os*), 723
WEXITSTATUS() (*dans le module os*), 725
wfile (*attribut http.server.BaseHTTPRequestHandler*), 1504
wfile (*attribut socketserver.DatagramRequestHandler*), 1498
whatis (*pdb command*), 1882
when() (*méthode asyncio.Timeout*), 1075
when() (*méthode asyncio.TimerHandle*), 1126
where (*pdb command*), 1880
which() (*dans le module shutil*), 501
whichdb() (*dans le module dbm*), 531
while
    statement, 37
While (*classe dans ast*), 2120
whitespace (*attribut shlex.shlex*), 1616
whitespace (*dans le module string*), 122
whitespace_split (*attribut shlex.shlex*), 1616
Widget (*classe dans tkinter.ttk*), 1644
--width
    option de ligne de commande calendar, 258
width (*attribut sys.hash_info*), 1942
width (*attribut textwrap.TextWrapper*), 170
width() (*dans le module turtle*), 1585

WIFCONTINUED() (*dans le module os*), 724
WIFEXITED() (*dans le module os*), 725
WIFSIGNALED() (*dans le module os*), 725
WIFSTOPPED() (*dans le module os*), 725
win32_edition() (*dans le module platform*), 802
win32_is_iot() (*dans le module platform*), 802
win32_ver() (*dans le module platform*), 802
WinDLL (*classe dans ctypes*), 833
window manager (*widgets*), 1628
window() (*méthode curses.panel.Panel*), 954
window_height() (*dans le module turtle*), 1601
window_width() (*dans le module turtle*), 1601
Windows ini file, 620
WindowsError, 114
WindowsPath (*classe dans pathlib*), 457
WindowsProactorEventLoopPolicy (*classe dans asyncio*), 1149
WindowsRegistryFinder (*classe dans importlib.machinery*), 2075
WindowsSelectorEventLoopPolicy (*classe dans asyncio*), 1149
winerror (*attribut OSError*), 111
WinError() (*dans le module ctypes*), 841
WINFUNCTYPE() (*dans le module ctypes*), 837
winreg
    module, 2186
WinSock, 1225
winsound
    module, 2194
winver (*dans le module sys*), 1954
With (*classe dans ast*), 2122
WITH_EXCEPT_START (*opcode*), 2170
with_hostmask (*attribut ipaddress.IPv4Interface*), 1546
with_hostmask (*attribut ipaddress.IPv4Network*), 1541
with_hostmask (*attribut ipaddress.IPv6Interface*), 1546
with_hostmask (*attribut ipaddress.IPv6Network*), 1544
with_name() (*méthode pathlib.PurePath*), 456
with_netmask (*attribut ipaddress.IPv4Interface*), 1546
with_netmask (*attribut ipaddress.IPv4Network*), 1541
with_netmask (*attribut ipaddress.IPv6Interface*), 1546
with_netmask (*attribut ipaddress.IPv6Network*), 1544
with_prefixlen (*attribut ipaddress.IPv4Interface*), 1545
with_prefixlen (*attribut ipaddress.IPv4Network*), 1541
with_prefixlen (*attribut ipaddress.IPv6Interface*), 1546
with_prefixlen (*attribut ipaddress.IPv6Network*), 1544
with_pymalloc() (*dans le module test.support*), 1850
with_segments() (*méthode pathlib.PurePath*), 456
with_stem() (*méthode pathlib.PurePath*), 456
with_suffix() (*méthode pathlib.PurePath*), 456
with_traceback() (*méthode BaseException*), 108
withitem (*classe dans ast*), 2122
WNOHANG (*dans le module os*), 723
WNOWAIT (*dans le module os*), 724
wordchars (*attribut shlex.shlex*), 1616
World Wide Web, 1413, 1443, 1452
wrap() (*dans le module textwrap*), 168
wrap() (*méthode textwrap.TextWrapper*), 172
wrap_bio() (*méthode ssl.SSLContext*), 1211
wrap_future() (*dans le module asyncio*), 1131
wrap_socket() (*méthode ssl.SSLContext*), 1210
wrapper() (*dans le module curses*), 930
WrapperDescriptorType (*dans le module types*), 302
wraps() (*dans le module functools*), 436
WRITABLE (*attribut inspect.BufferFlags*), 2044
WRITABLE (*dans le module _tkinter*), 1632
writable() (*méthode bz2.BZ2File*), 575
writable() (*méthode io.IOBase*), 735
WRITE (*attribut inspect.BufferFlags*), 2044
write() (*dans le module os*), 684
write() (*dans le module turtle*), 1589
write() (*méthode asyncio.StreamWriter*), 1088
write() (*méthode asyncio.WriteTransport*), 1137
write() (*méthode codecs.StreamWriter*), 196
write() (*méthode code.InteractiveInterpreter*), 2054
write() (*méthode configparser.ConfigParser*), 637
write() (*méthode email.generator.BytesGenerator*), 1263
write() (*méthode email.generator.Generator*), 1264
write() (*méthode io.BufferedIOBase*), 737
write() (*méthode io.BufferedWriter*), 739
write() (*méthode io.RawIOBase*), 736
write() (*méthode io.TextIOBase*), 740
write() (*méthode mmap.mmap*), 1247
write() (*méthode sqlite3.Blob*), 555
write() (*méthode ssl.MemoryBIO*), 1221
write() (*méthode ssl.SSLSocket*), 1202
write() (*méthode xml.etree.ElementTree.ElementTree*), 1369
write() (*méthode zipfile.ZipFile*), 589
write_byte() (*méthode mmap.mmap*), 1247
write_bytes() (*méthode pathlib.Path*), 463
write_docstringdict() (*dans le module turtle*), 1605
write_eof() (*méthode asyncio.StreamWriter*), 1088
write_eof() (*méthode asyncio.WriteTransport*), 1138
write_eof() (*méthode ssl.MemoryBIO*), 1221
write_history_file() (*dans le module readline*), 176
write_results() (*méthode trace.CoverageResults*), 1900
write_text() (*méthode pathlib.Path*), 463
write_through (*attribut io.TextIOWrapper*), 741
writeframes() (*méthode wave.Wave_write*), 1551
writeframesraw() (*méthode wave.Wave_write*), 1551
writeheader() (*méthode csv.DictWriter*), 619
writelines() (*méthode asyncio.StreamWriter*), 1088
writelines() (*méthode asyncio.WriteTransport*), 1137

writelines() (*méthode codecs.StreamWriter*), 196
writelines() (*méthode io.IOBase*), 735
writepy() (*méthode zipfile.PyZipFile*), 592
writer() (*dans le module csv*), 614
writerow() (*méthode csv.csvwriter*), 619
writerows() (*méthode csv.csvwriter*), 619
writestr() (*méthode zipfile.ZipFile*), 590
WriteTransport (*classe dans asyncio*), 1135
writev() (*dans le module os*), 684
writexml() (*méthode xml.dom.minidom.Node*), 1386
WrongDocumentErr, 1383
wsgi_file_wrapper (*attribut wsgiref.handlers.BaseHandler*), 1423
wsgi_multiprocess (*attribut wsgiref.handlers.BaseHandler*), 1422
wsgi_multithread (*attribut wsgiref.handlers.BaseHandler*), 1422
wsgi_run_once (*attribut wsgiref.handlers.BaseHandler*), 1422
WSGIApplication (*dans le module wsgiref.types*), 1424
WSGIEnvironment (*dans le module wsgiref.types*), 1424
wsgiref
    module, 1416
wsgiref.handlers
    module, 1421
wsgiref.headers
    module, 1418
wsgiref.simple_server
    module, 1419
wsgiref.types
    module, 1424
wsgiref.util
    module, 1416
wsgiref.validate
    module, 1420
WSGIRequestHandler (*classe dans wsgiref.simple_server*), 1420
WSGIServer (*classe dans wsgiref.simple_server*), 1419
wShowWindow (*attribut subprocess.STARTUPINFO*), 1040
WSTOPPED (*dans le module os*), 723
WSTOPSIG() (*dans le module os*), 725
wstring_at() (*dans le module ctypes*), 841
WTERMSIG() (*dans le module os*), 725
WUNTRACED (*dans le module os*), 723
WWW, 1413, 1443, 1452
    server, 1502

X

-x
    option de ligne de commande compileall, 2156
X (*dans le module re*), 141
X509 certificate, 1215
X_OK (*dans le module os*), 687
xatom() (*méthode imaplib.IMAP4*), 1481
XATTR_CREATE (*dans le module os*), 712
XATTR_REPLACE (*dans le module os*), 712
XATTR_SIZE_MAX (*dans le module os*), 711
xcor() (*dans le module turtle*), 1583
xdrlib
    module, 2225
XHTML, 1347
XHTML_NAMESPACE (*dans le module xml.dom*), 1375
xml
    module, 1353
XML() (*dans le module xml.etree.ElementTree*), 1364
XML_ERROR_ABORTED (*dans le module xml.parsers.expat.errors*), 1412
XML_ERROR_AMPLIFICATION_LIMIT_BREACH (*dans le module xml.parsers.expat.errors*), 1412
XML_ERROR_ASYNC_ENTITY (*dans le module xml.parsers.expat.errors*), 1410
XML_ERROR_ATTRIBUTE_EXTERNAL_ENTITY_REF (*dans le module xml.parsers.expat.errors*), 1410
XML_ERROR_BAD_CHAR_REF (*dans le module xml.parsers.expat.errors*), 1410
XML_ERROR_BINARY_ENTITY_REF (*dans le module xml.parsers.expat.errors*), 1410
XML_ERROR_CANT_CHANGE_FEATURE_ONCE_PARSING (*dans le module xml.parsers.expat.errors*), 1411
XML_ERROR_DUPLICATE_ATTRIBUTE (*dans le module xml.parsers.expat.errors*), 1410
XML_ERROR_ENTITY_DECLARED_IN_PE (*dans le module xml.parsers.expat.errors*), 1411
XML_ERROR_EXTERNAL_ENTITY_HANDLING (*dans le module xml.parsers.expat.errors*), 1411
XML_ERROR_FEATURE_REQUIRES_XML_DTD (*dans le module xml.parsers.expat.errors*), 1411
XML_ERROR_FINISHED (*dans le module xml.parsers.expat.errors*), 1412
XML_ERROR_INCOMPLETE_PE (*dans le module xml.parsers.expat.errors*), 1411
XML_ERROR_INCORRECT_ENCODING (*dans le module xml.parsers.expat.errors*), 1410
XML_ERROR_INVALID_ARGUMENT (*dans le module xml.parsers.expat.errors*), 1412
XML_ERROR_INVALID_TOKEN (*dans le module xml.parsers.expat.errors*), 1410
XML_ERROR_JUNK_AFTER_DOC_ELEMENT (*dans le module xml.parsers.expat.errors*), 1410
XML_ERROR_MISPLACED_XML_PI (*dans le module xml.parsers.expat.errors*), 1410
XML_ERROR_NO_BUFFER (*dans le module xml.parsers.expat.errors*), 1412
XML_ERROR_NO_ELEMENTS (*dans le module xml.parsers.expat.errors*), 1410
XML_ERROR_NO_MEMORY (*dans le module xml.parsers.expat.errors*), 1410
XML_ERROR_NOT_STANDALONE (*dans le module xml.parsers.expat.errors*), 1411
XML_ERROR_NOT_STARTED (*dans le module xml.parsers.expat.errors*), 1412
XML_ERROR_NOT_SUSPENDED (*dans le module*

xml.parsers.expat.errors), 1411
XML_ERROR_PARAM_ENTITY_REF (*dans le module xml.parsers.expat.errors*), 1410
XML_ERROR_PARTIAL_CHAR (*dans le module xml.parsers.expat.errors*), 1410
XML_ERROR_PUBLICID (*dans le module xml.parsers.expat.errors*), 1411
XML_ERROR_RECURSIVE_ENTITY_REF (*dans le module xml.parsers.expat.errors*), 1410
XML_ERROR_RESERVED_NAMESPACE_URI (*dans le module xml.parsers.expat.errors*), 1412
XML_ERROR_RESERVED_PREFIX_XML (*dans le module xml.parsers.expat.errors*), 1412
XML_ERROR_RESERVED_PREFIX_XMLNS (*dans le module xml.parsers.expat.errors*), 1412
XML_ERROR_SUSPEND_PE (*dans le module xml.parsers.expat.errors*), 1412
XML_ERROR_SUSPENDED (*dans le module xml.parsers.expat.errors*), 1411
XML_ERROR_SYNTAX (*dans le module xml.parsers.expat.errors*), 1411
XML_ERROR_TAG_MISMATCH (*dans le module xml.parsers.expat.errors*), 1411
XML_ERROR_TEXT_DECL (*dans le module xml.parsers.expat.errors*), 1411
XML_ERROR_UNBOUND_PREFIX (*dans le module xml.parsers.expat.errors*), 1411
XML_ERROR_UNCLOSED_CDATA_SECTION (*dans le module xml.parsers.expat.errors*), 1411
XML_ERROR_UNCLOSED_TOKEN (*dans le module xml.parsers.expat.errors*), 1411
XML_ERROR_UNDECLARING_PREFIX (*dans le module xml.parsers.expat.errors*), 1411
XML_ERROR_UNDEFINED_ENTITY (*dans le module xml.parsers.expat.errors*), 1411
XML_ERROR_UNEXPECTED_STATE (*dans le module xml.parsers.expat.errors*), 1411
XML_ERROR_UNKNOWN_ENCODING (*dans le module xml.parsers.expat.errors*), 1411
XML_ERROR_XML_DECL (*dans le module xml.parsers.expat.errors*), 1411
XML_NAMESPACE (*dans le module xml.dom*), 1375
xmlcharrefreplace
    error handler's name, 192
xmlcharrefreplace_errors() (*dans le module codecs*), 193
XmlDeclHandler() (*méthode xml.parsers.expat.xmlparser*), 1406
xml.dom
    module, 1374
xml.dom.minidom
    module, 1384
xml.dom.pulldom
    module, 1389
xml.etree.ElementInclude
    module, 1366
xml.etree.ElementTree
    module, 1354
XMLFilterBase (*classe dans xml.sax.saxutils*), 1399
XMLGenerator (*classe dans xml.sax.saxutils*), 1398
XMLID() (*dans le module xml.etree.ElementTree*), 1364
XMLNS_NAMESPACE (*dans le module xml.dom*), 1375
XMLParser (*classe dans xml.etree.ElementTree*), 1371
xml.parsers.expat
    module, 1403
xml.parsers.expat.errors
    module, 1410
xml.parsers.expat.model
    module, 1409
XMLParserType (*dans le module xml.parsers.expat*), 1403
XMLPullParser (*classe dans xml.etree.ElementTree*), 1373
XMLReader (*classe dans xml.sax.xmlreader*), 1399
xmlrpc
    module, 1521
xmlrpc.client
    module, 1521
xmlrpc.server
    module, 1528
xml.sax
    module, 1391
xml.sax.handler
    module, 1392
xml.sax.saxutils
    module, 1398
xml.sax.xmlreader
    module, 1399
xor() (*dans le module operator*), 439
xview() (*méthode tkinter.ttk.Treeview*), 1654

Y

ycor() (*dans le module turtle*), 1584
year
    option de ligne de commande calendar, 258
year (*attribut datetime.date*), 214
year (*attribut datetime.datetime*), 221
Year 2038, 744
yeardatescalendar() (*méthode calendar.Calendar*), 251
yeardays2calendar() (*méthode calendar.Calendar*), 251
yeardayscalendar() (*méthode calendar.Calendar*), 251
YES (*dans le module tkinter.messagebox*), 1638
YESEXPR (*dans le module locale*), 1564
YESNO (*dans le module tkinter.messagebox*), 1639
YESNOCANCEL (*dans le module tkinter.messagebox*), 1639
Yield (*classe dans ast*), 2132
YIELD_VALUE (*opcode*), 2169
YieldFrom (*classe dans ast*), 2132
yiq_to_rgb() (*dans le module colorsys*), 1552
yview() (*méthode tkinter.ttk.Treeview*), 1654

Z

z
    in string formatting, 126
z85decode() (*dans le module base64*), 1341
z85encode() (*dans le module base64*), 1341
ZeroDivisionError, 114
zfill() (*méthode bytearray*), 78
zfill() (*méthode bytes*), 78
zfill() (*méthode str*), 62
zip()
    built-in function, 32
ZIP_BZIP2 (*dans le module zipfile*), 585
ZIP_DEFLATED (*dans le module zipfile*), 585
zip_longest() (*dans le module itertools*), 420
ZIP_LZMA (*dans le module zipfile*), 585
ZIP_STORED (*dans le module zipfile*), 585
zipapp
    module, 1924
zipfile
    module, 584
ZipFile (*classe dans zipfile*), 586
zipimport
    module, 2057
zipimporter (*classe dans zipimport*), 2058
ZipImportError, 2058
ZipInfo (*classe dans zipfile*), 585
zlib
    module, 567
ZLIB_RUNTIME_VERSION (*dans le module zlib*), 570
ZLIB_VERSION (*dans le module zlib*), 570
zoneinfo
    module, 245
ZoneInfo (*classe dans zoneinfo*), 247
ZoneInfoNotFoundError, 250
zscore() (*méthode statistics.NormalDist*), 405